

 WILEY

OS/2TM PRESENTATION MANAGER PROGRAMMING



Paul W. Cheatham
David E. Reich
Robert F. G. Robinson

*Covers
Version
1.2*

Foreword by Tommy Steele
Director, Boca Programming Center

OS/2® Presentation Manager Programming

Related titles of interest from Wiley:

OS/2 Database Manager: A Developer's Guide, Fosdick

OS/2: Features, Functions and Applications, Krantz, Mizell, and Williams

OS/2: A Business Perspective, Conklin

IBM PS/2: A Business Perspective, Revised Edition, Hoskins

IBM PS/2 User's Reference Manual, Held

Desktop Communications: IBM PC, PS/2, and Compatibles, Honig and Hoover

The New DOS 4.0, Christopher, Feigenbaum, and Saliga

DOS 4.0 Reference, Christopher, Feigenbaum, and Saliga

OS/2[®] PRESENTATION MANAGER PROGRAMMING

Paul W. Cheatham
David E. Reich
Robert F. G. Robinson



John Wiley & Sons, Inc.

New York

Chichester

Brisbane

Toronto

Singapore

Editor: Therese A. Zak
Managing Editor: Ruth Greif
Editing, Design, and Production: Impressions Publishing Services, Inc.

This book is not sponsored in any part or in any manner by IBM. IBM does not endorse or represent the accuracy or appropriateness of any information contained herein. In addition, this book is not intended to replace IBM product documentation or personnel in determining the specifications and capabilities of the included products. You are responsible for choice of all configurations and applications of computer hardware and software. You should discuss these choices with the proper IBM representative.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional service. If legal advice or other expert assistance is required, the services of a competent professional person should be sought. FROM A DECLARATION OF PRINCIPLES JOINTLY ADOPTED BY A COMMITTEE OF THE AMERICAN BAR ASSOCIATION AND A COMMITTEE OF PUBLISHERS.

Copyright © 1990 by John Wiley & Sons, Inc.

All rights reserved. Published simultaneously in Canada.

Reproduction or translation of any part of this work beyond that permitted by section 107 or 108 of the 1976 United States Copyright Act without the permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to the Permission Department, John Wiley & Sons, Inc.

Library of Congress Cataloging-in-Publication Data

Cheatham, Paul W.

OS/2 presentation manager programming / Paul W. Cheatham, David E.

Reich, Robert F. G. Robinson.

p. cm.

Includes bibliographical references.

ISBN 0-471-50897-7

1. OS/2 (Computer operating system) 2. Presentation manager
(Computer program) I. Reich, David E. II. Robinson, Robert F. G.

III. Title.

QA76.76.063C457 1989

005.4'469—dc20

89-16736
CIP

Printed in the United States of America

92 93 10 9 8 7 6 5 4 3 2

Printed and bound by Malloy Lithographing, Inc.

*This book is dedicated to our wives—Stacy, Ann, and Jackie—
and our families for their patience and understanding.
Without their support, we could not have completed this book.*

Contents

Section One—Evolution of DOS and OS/2 Operating Systems

I DOS 5

IBM Personal Computer DOS Version 1.0	5
IBM Personal Computer DOS Version 1.1	6
IBM Personal Computer DOS Version 2.0	6
IBM Personal Computer DOS Version 2.1	7
IBM Personal Computer DOS Version 3.0	7
IBM Personal Computer DOS Version 3.1	8
IBM Personal Computer DOS Version 3.2	8
IBM DOS Version 3.3	9
IBM DOS Version 4.0	10
Summary	10

2 OS/2 Standard Edition 11

Features of OS/2 Version 1.0	11
Operating Modes	13
DOS Mode	13
OS/2 Mode	13

CONTENTS

Types of Applications	14
OS/2 Applications	14
DOS Applications	15
FAPI Applications	15
Protection	15
Multitasking	15
Memory Management	16
Dynamic Link Libraries and Subsystems	16
Interprocess Communication	17
Program Selector	17
Device Drivers	17
OS/2 Standard Edition Version 1.1	18
Kernel Enhancements	19
Presentation Manager	19
Windows	20
Graphics	20
User Interface	20
Device Drivers	22
Toolkit	22
OS/2 Standard Edition Version 1.2	22
Presentation Manager Enhancements	22
High Performance File System	23
Dialog Manager	23
Information Presentation Facility	23
Compatibility with DOS 4.0	24
Online Documentation	24
Windowed System Editor	24
IBM FORTRAN/2 Version 1.02 and IBM COBOL/2 Language Support	24
Summary	24

Section Two—Using OS/2 Version 1.2

3 Operating Environments 31

DOS Compatibility Box	31
OS/2 Full-Screen Mode	32
Presentation Manager User Shell	32
Windows	33
Extended Sessions	35
Summary	37

4 User Features in OS/2 38

Using the Mouse and Keyboard 38

The Mouse 38

The Keyboard 39

Using Windows 39

Selecting Menu Items 40

Moving Windows 42

Sizing Windows 42

Scrolling Data in Windows 44

Obtaining Online Help 45

Using Dialog Boxes 46

Clipboard Functions 47

The User Shell 48

Desktop Manager 48

Task List 50

File System 51

Control Panel 51

Print Manager 52

Summary 53

Section Three—Programming Concepts

5 OS/2 Base System Features 59

The Multitasking Model 59

OS/2 Protection 61

Device Drivers and Device Independence 66

Handles 69

APIs 70

Dynamic Linking 70

Subsystems 72

HPFS and IFS 73

Advanced Utilities 73

Summary 73

6 OS/2 Presentation Manager Functions 75

Types of Applications 75

What Is SAA, Anyway? 77

Presentation Manager Device Drivers 77

CONTENTS

The Presentation Manager Session	81
APIs	81
Win Calls	82
Gpi Calls	82
Windowing	83
Messages and Queues	84
Resource Management	85
Hooks	86
Graphic Object Manipulation	88
Clipboard	89
Dynamic Data Exchange	90
User Shell Program Interaction	91
Summary	91
7 Building an Application	94
Source Code Preparation	94
INCLUDE Files	95
Naming Conventions	95
Compiling and Linking	99
Memory Models	100
Module Definition File	100
Resource Compiler	102
MAKE Utility	102
Summary	104
8 Message Architecture	105
Message Structure	106
Threads and Queues	107
Message Flow	108
Message Priority	111
Summary	113
9 Presentation Manager Program Structure	114
Main Processing Routine	114
Window Procedure	116
Summary	118

Section Four—Window Programming

10 Presentation Manager Versus Traditional Programming 123

- Sample VioWrtTTY Program 123
- Sample Presentation Manager Program 124
- Sample Program Window Procedure 132
- Summary 134

11 Window Programming Concepts 135

- Object-Oriented Programming 135
 - Presentation Manager and Object-Oriented Programming 137
- Window Relationships 138
 - Parent-Child Relationship 138
 - Owner-Owned Relationship 141
- Window Classes 142
 - Public Classes 142
 - Private Classes 143
- Basic Versus Standard Windows 146
 - The Basic Window 147
 - The Standard Window 148
 - The WM_CREATE Message 151
 - Window Presentation and Behavior 152
 - Styles 152
 - Painting 154
- Controlling User Input 155
 - Focus 155
 - Capture 157
- Communicating with Windows 158
 - Identifying Windows 158
 - Sending Messages 160
 - Posting Messages 161
- Summary 161

12 Messages 163

- Building a Window Procedure 164
- Handling Messages 167
 - Message Return Values 167
 - WM_COMMAND 168

WM_CONTROL	171
WM_PAINT	173
Mouse Messages	174
Keyboard Messages (WM_CHAR)	176
Summary	179
13 Dialogs	180
Communicating with Dialog Controls	181
Modality	182
Modeless Dialogs	183
Modal Dialogs	183
Dialog Controls	185
Radiobuttons	186
Pushbuttons	188
Check Boxes	189
Listboxes	190
Entry Fields	194
Combo Boxes	197
Scroll Bars	198
Multiline Edit Controls	204
Dialog Procedure	205
Message Boxes	208
Summary	211
14 User Menus	212
Creating Menus	213
Communicating with the Menu	219
Detecting Menu Selection	219
Controlling the Menu	221
Summary	223

Section Five—Graphics Programming Interface (GPI)

15 Device Contexts	229
Screen Device Context	231
Metafile Device Context	231
Memory Device Context	232

Printer Device Context	232
Summary	234

16 Presentation Spaces 235

Normal Presentation Space	236
Micro Presentation Space	236
Cached Micro Presentation Space	237
Creating Normal and Micro Presentation Spaces	238
Pels	240
Metric Units	240
Arbitrary Units	241
Summary	242

17 Graphics Primitives 244

Arc Primitives	245
Area Primitives	251
Characters	255
Character Angle	256
Character Box	256
Character Shear	256
Images	257
Line Primitives	257
Markers	261
Summary	262

18 Graphics Character Support 263

Displaying Text Strings	263
VIO Text Writing	264
Window Text Drawing	265
Graphics Text Functions	267
Changing Fonts	272
Creating Markers	278
Creating Patterns	279
Bitmap Patterns	279
Font Patterns	280
Summary	281

19 Graphics Segments 282

Modes	282
Draw Mode	282

Retain Mode	283
Draw and Retain Mode	283
Creating a Segment	284
Editing a Segment	289
Copying Segment Data	293
Segment Attributes	293
Summary	295
20 Transforms	296
Coordinate Spaces	296
Segment Transforms	298
Model Transforms	298
Viewing Transforms	299
Device Space Transforms	300
Transformation Matrix	300
Translation	300
Scaling	301
Rotation	301
Summary	302
21 Storing and Printing Application Graphics	303
Bitmaps	303
Bitmap Header	303
Bitmap Data	304
Creating a Bitmap	304
Loading a Bitmap	307
Metafiles	308
Using an Existing Metafile	310
Printing Graphics	310
Simple Text Printing	311
Direct Printing	311
Queued Printing	314
Summary	318

Section Six—Advanced Topics

22 Application Data Transfer 323

Clipboard	324
Rendering	325

Copy	325
Cut	326
Paste	327
Delayed Rendering	327
Copy	328
Cut	328
Dynamic Data Exchange	329
Summary	331

23 Subclassing Windows 332

What Is Subclassing?	332
Performing the Subclass	333
Unsubclassing	333
Using the Subclass Procedure	333
Watch That Return Chain!	337
Multiple Subclasses	338
Summary	339

24 Placing Resources in Dynamic Link Libraries 340

Creating a Dynamic Link Library Stub	340
Creating a Font Library	342
Creating a Resource Library	343
Summary	344

25 Application Debugging Tips 345

Free Style Debugging	346
Hardware Debugging	347
Symbolic Debugging	349
Summary	350

26 Advanced VIO Programming 351

27 Window Templates 370

Creating and Using Window Templates	370
Summary	374

A Application Programming Interfaces 375

B Structures 503

C	Variable Prefixes	538
D	Restricted KBD, MOU, and VIO Calls	543
E	Include File Conditional Selection	545
F	Helper Macros	548
	Index	551

Foreword

With its introduction of the Personal System/2® family of products, IBM® brought significant new capabilities to the personal computing environment. The OS/2 operating system gives application developers access to the additional functions and enhanced ease-of-use characteristics that the PS/2 hardware provides. Large memory support, multitasking capability, windowing, and IBM DOS compatibility make the OS/2 operating system both powerful and versatile. By providing a state-of-the-art graphical interface for the windowing environment, the Presentation Manager enables programmers to deliver an unprecedented array of applications to the users of their products.

Because the Presentation Manager is such a powerful tool for enhancing usability, there is much to learn about its features and functions. Few people have had the opportunity to develop an in-depth knowledge of the Presentation Manager in day-to-day applications. This book shares the insights of three IBM software engineers who have worked with the Presentation Manager as it evolved from a set of specifications through design changes and testing to a final product. They have tested the product, consulted with thousands of early users, and provided an interface between users and IBM software developers. They know the Presentation Manager as programmers who have written applications and as users of programs written by others. This combination of experience and knowledge results in a “real-world” perspective that other programmers should appreciate.

I congratulate the authors on compiling a relevant, readable guide.

—Tommy Steele, Director, Boca Programming Center

Preface

OS/2 is an impressive achievement on the part of the IBM and Microsoft development teams. It is an operating system that overcomes many of the limitations of DOS, adding a friendly user interface (the Presentation Manager) as well as providing functions to programs to make them do more, perform better, and even use less memory with dynamic linking.

Along with all this functionality comes a great set of programming challenges. This book will help you overcome these hurdles.

Throughout this book we concentrate on the fundamentals. The OS/2 Presentation Manager is a complex system, and mastering the basic principles unlocks virtually unlimited programming possibilities. We have strategically placed program code to illustrate these fundamentals. You can use the code as the building blocks for all your programs.

This book is organized into six sections:

- How DOS and OS/2 evolved into what they are today
- Using the OS/2 system
- Highlights of the features of OS/2
- Presentation Manager window programming
- The Graphics Programming Interface
- A series of more advanced topics

PREFACE

In addition, the appendixes provide comprehensive lists of API function prototypes, data structures, and macros as well as descriptions of the different categories of functions.

This book is designed to give you an overall picture of OS/2. We dig into many pieces of the system and explain how and why things work. We cannot stress enough how important these concepts are. Through our experience working with programmers and users, both seasoned and novice, we've found that the skills presented here should be the foundation of your Presentation Manager repertoire.

We've tried to make this book very instructional and pleasant to read. We hope you enjoy it.

Acknowledgments

We would like to thank the talented IBM and Microsoft management teams, designers, developers, testers, planners, and support personnel, without whose hard work and dedication OS/2 would not be a reality.

Special thanks go to Keith Bernstein, Pete Woods, Sam Detweiler, Chris King, Mike Edwards, and Carol Ziemba for their valuable insight and comments on the technical content and structure of this book. In addition, we would like to thank our managers, Susan Macke, Ken Bauler, and Hal Opdyke for their time spent reviewing the text.

Trademark Acknowledgments

Codeview, Microsoft, and Microsoft C version 5.1 are trademarks of Microsoft Corporation.

IBM C/2 and Operating System/2 are trademarks and IBM Personal Computer AT, IBM Personal Computer XT, OS/2, Personal System/2, and PS/2 are registered trademarks of IBM Corporation.

Intel is a registered trademark of Intel Corporation.

UNIX is a trademark of AT&T Corporation.

SECTION ONE

EVOLUTION OF DOS AND OS/2 OPERATING SYSTEMS

On August 12, 1981, International Business Machines Corporation entered a new era with the announcement of the IBM® Personal Computer®. The IBM Personal Computer Diskette Software (DOS Diskette) was touted as “providing additional device and system software support.”

The first chapter of this section traces the evolution of DOS, discussing the highlights of each release.

Chapter 2 covers the evolution of IBM Operating System/2®, which evolved from DOS. The characteristics of the OS/2® operating system and some of the reasons for having a new operating system are explained. The second release of the OS/2 operating system introduced the Presentation Manager facility. The Presentation Manager facility is the focus of this book.

The discussion of the latest release of OS/2, Version 1.2, covers the enhancements to the operating system, including the specific enhancements to the Presentation Manager facility.

I

DOS

Soon after IBM introduced the IBM Personal Computer Diskette Software (DOS Diskette), it became known as PC DOS. PC DOS, designed around the Intel® 8088 microprocessor, became the operating system on which many applications were developed over the years. As new hardware was developed and users became more sophisticated, new releases of the operating system were announced. Some of the releases provided support for new hardware. Other releases contained significant new functions and/or enhancements to existing functions. Application developers and users were able to get more out of their systems with each new release.

This chapter takes a brief look at each release to see how the PC DOS software evolved. You will see that the memory required to hold the operating system has increased, as well as the memory required to run the operating system. The major enhancements and new functions introduced in each release will be highlighted.

IBM Personal Computer DOS Version 1.0

IBM Personal Computer DOS Version 1.0 (PC DOS) started as the IBM Personal Computer Diskette Software (DOS Diskette). It became available in October 1981. This operating system was said to provide a high-level interface between a program and its hardware environment. It had support for one or more 5.25-inch 160KB single-sided diskette drives with both sequential and random access of files.

Basic diskette utilities provided the capability to display the directory and to display, rename, copy, erase, and compare files. Predefined job streams (called *batch files*)

containing a series of DOS commands could be defined and executed. An extension to the BASIC interpreter provided diskette operations and enhancements to the display graphics, speaker, light pen, and joystick support. There was diskette I/O support for the Pascal compiler. Files could be created and modified with a line editor that was similar to those in other IBM operating systems. A debug utility program was included to help in debugging programs. The system contained a linkage editor to convert language-compiler-relocatable modules to executable-load modules.

DOS Version 1.0 required a minimum of one diskette drive. The operating system resided on a diskette and had to be loaded into memory from drive A. Approximately 12KB of memory was required for loading DOS, with a minimum of 32KB of memory required to execute programs. The basic PC had a 4.77-MHz Intel 8088 microprocessor with 16KB of memory and a single 160KB diskette drive. That may not sound like much in today's computing environment, but at the time the PC was announced it was considered a giant step forward for personal computing. The IBM PC and DOS gained instant success as IBM's entry into the personal computer market.

IBM Personal Computer DOS Version 1.1

In the spring of 1982, IBM announced DOS Version 1.1, with support for the new IBM 5.25-inch 320KB diskette drive. This release of DOS included all the features of Version 1.0, with added support for the new double-sided diskettes. The horizon had begun to expand.

IBM Personal Computer DOS Version 2.0

The introduction of fixed disks for the IBM Personal Computer and the IBM Personal Computer XT™ created a need for an enhanced operating system to support this new hardware. Announced on March 3, 1983, IBM Personal Computer DOS Version 2.0 required approximately 12KB more memory to load than DOS 1.1 and featured fixed disk support for up to two fixed disks, as well as support for one or more 5.25-inch diskette drives. Other features included the capability to start DOS either from diskette or fixed disk, BACKUP and RESTORE commands in support of the fixed disk, a new hierarchical directory structure, input/output (I/O) redirection, support for conditional (IF-THEN-ELSE) logic in batch files, and a background file print utility permitting file printing simultaneously with other computing activity. Diskette capacity was increased to 180KB for single-sided diskettes and to 360KB for double-sided diskettes.

Several significant changes took place with the introduction of the new directory structure. In the previous versions of DOS, there had been a single directory with a

limit of 64 files with access through file control blocks (FCBs). The new hierarchical directory structure allowed the main directory, called the *root directory*, to contain both files and directory structures, called *subdirectories*, within it. Each of these subdirectories could, in turn, contain files and subdirectories of its own. Files were now accessed with file handles rather than FCBs. The root directory was limited to 256 files, but there was no limit to the number of files in the subdirectories or to the number of levels of subdirectories. There were commands to support the new directory structure: display the directory structure, create subdirectories, and remove subdirectories.

DOS Version 2.0 required a minimum of one diskette drive and 64KB memory, with a recommended minimum of 128KB if a fixed disk was installed. The size of the operating system had started to grow as new hardware support and new user functions were added. The addition of fixed disk support was a milestone.

IBM Personal Computer DOS Version 2.1

DOS Version 2.1, announced on November 1, 1983, was the first version of DOS to support the IBM PCjr™. This hardware support was added without increasing the memory requirements of the operating system.

IBM Personal Computer DOS Version 3.0

On August 14, 1984, IBM announced DOS 3.0, which became available in September 1984. DOS 3.0 provided all the functions contained in DOS 2.1, plus enhancements for support of the new IBM Personal Computer AT® system hardware. The IBM PC AT was the first personal computer made by IBM to use the Intel 80286 microprocessor. DOS 3.0 used only the “real” mode of the 80286 and ran the machine as a “fast 8086.”

At least 36KB of memory was required to load the new operating system, 12KB more than DOS 2.1. The features of DOS 3.0 included virtual disk support to allow the use of part of memory as a virtual disk (the amount could exceed 1MB), a full range of file-sharing features, the ability to restrict access to all or part of a file when the file is opened in a shared mode (block locking), improved background printing, which supports path specifications and an internal programming interface, enhanced error recovery with additional error reporting facilities, support for IBM Personal Computer AT hardware (with 1.2MB diskette drives, nonvolatile timer, and a larger fixed disk), a screen dump utility with support for additional display interfaces and printers, and a large linker supporting up to 1MB.

Also included was code to be used in DOS 3.1, which was announced at the same time. DOS 3.0 was an interim product, which was replaced quickly by DOS 3.1.

DOS 3.0 required a minimum of 96KB to 128KB memory with 128KB recommended for a system unit with fixed disk. Memory above 1MB was accessible, and this helped speed up operations on the system, because users could load often-used programs onto the virtual disk. Using programs from virtual disks saved time compared with accessing the program files on a physical disk. They still could not be executed from memory above 1MB, however.

IBM Personal Computer DOS Version 3.1

Announced simultaneously with DOS 3.0 on August 14, 1984, DOS 3.1 was designed to replace DOS 3.0 and provide support for IBM PC Network. Availability was scheduled for the first quarter of 1985. DOS 3.1 had the same storage requirements as DOS 3.0 and provided the following network support enhancements: a redirector (the means by which a user could access printer and direct access storage devices [DASD] located on other computers on the IBM PC Network), new interrupt 21H functions (additional function calls providing support for the IBM PC Network environment), extended error recovery (additional error reporting facilities to enhance support in the IBM PC Network environment), the JOIN utility (allowed splicing of directories), and the SUBST utility (to substitute a virtual drive name in a path name).

The network environment required significant new support from the operating system, but it gave the user capabilities for sharing and distributing data and files that had not been available previously. There were many subtle changes that had to be made in DOS to support a network environment. Problems that never occurred in a stand-alone environment occurred frequently in a network environment. The operating system was growing to meet the needs of the user.

IBM Personal Computer DOS Version 3.2

With the introduction of new hardware, IBM announced DOS 3.2 in the spring of 1986. The operating system itself required 44KB of memory, up 8KB from DOS 3.1. DOS 3.2 provided all of the functions contained in DOS 3.1 plus support for the new 720KB 3.5-inch diskette drives, the IBM Token-Ring Network, and the IBM PC Convertible™.

Two new commands, REPLACE and XCOPY, aided in diskette and disk management. REPLACE allowed replacement of all occurrences of a file on a disk to facilitate installation of new versions of DOS or application programs. XCOPY allowed copying of files from more than one subdirectory where the source and target drives were of different densities, making complete or periodic backups easier.

The system provided logical drive support that allowed a physical drive to be referenced by more than one device letter. There was also a new option allowing the DOS environment area to be expanded to 32KB. This enabled the user to specify more extensive PATH commands, PROMPT commands, and SET commands without running out of environment.

The FORMAT command was enhanced to require the specification of a drive letter. In previous versions, if no drive letter was specified, the current drive was used as the default. It was very easy to accidentally reformat the fixed disk in these versions. The new requirement made that accident less likely to occur. Device drivers now contained the formatting parameters for a specific device.

The range of hardware supported by the operating system included the IBM PC Convertible with a minimum of 256KB, the IBM PCjr with a minimum of 128KB, the IBM Personal Computer with a minimum of 96KB (128KB recommended for a fixed disk), the IBM Personal Computer XT with a minimum of 128KB, the IBM Portable Personal Computer™ with a minimum of 256KB, and the IBM Personal Computer AT with a minimum of 256KB.

The user was required to have at least one of the following: a 360KB diskette drive, a 720KB 3.5-inch diskette drive, or a 1.2MB diskette drive.

The range of hardware had expanded and the functions that users required had expanded. Yet DOS 3.2 was still a single-user, single-tasking operating system.

IBM DOS Version 3.3

On April 2, 1987, IBM announced DOS 3.3. The previous versions had all been called "IBM Personal Computer" operating systems, but that phrase was not included in the name of this version of DOS. The reason for this change was the introduction of a new generation of hardware, the IBM Personal System/2® family. The IBM Disk Operating System (DOS) now supported both the IBM Personal Computer family and the IBM Personal System/2 family. DOS 3.3 was compatible with DOS 3.2 and featured new commands, enhancements to old commands, performance enhancements, and new hardware support.

The three new commands in DOS 3.3 were FASTOPEN (support for file-name cache), CALL (to allow nested batch files), and APPEND (to allow access to applications and data files not contained in the current directory).

BACKUP, RESTORE, DATE, TIME, ATTRIB, and SYS were enhanced, and other enhancements increased the number of open files, sped up disk drive I/O, and provided a faster and more secure method of writing to a disk file in multiuser environments such as networks. There was new support for four asynchronous ports, 1.44MB diskettes, fixed disks greater than 32MB in partitioned mode, the new PS/2® Micro Channel®, and the Intel 80386 microprocessor in real mode.

The stage was being set for a relationship between DOS and a new, powerful operating system that would take advantage of the 80286- and 80386-based hardware. DOS was growing and adding functions, but it still supported only a single user and a single task.

IBM DOS Version 4.0

DOS Version 4.0 was announced on July 19, 1988. It was the first version of DOS to be released following the announcement of OS/2 Version 1.0. This version of DOS incorporated some of the features of OS/2. There was an attempt to make the user interaction as much alike as possible in the two systems. The DOS shell resembles the OS/2 Version 1.0 Program Selector.

The features of DOS 4.0 include a program warranty with defect service for one year after availability; support for fixed disks greater than 32MB in a nonpartitioned manner; a user shell providing program, file, and directory services as user-friendly alternatives to the command line; support for Expanded Memory Specification-capable memory adapters; and enhancements in video/graphics support to provide printing of graphics screens, additional video mode settings, and display of more than 25 lines of text. DOS 4.0 is the latest version of the IBM single-user, single-tasking operating system as of the writing of this book.

Summary

Since 1981 a tremendous number of DOS-based applications have been written. The memory requirements and the applications themselves have grown larger over the years, the result of complexity of the software's features as well as the increased amount of functions needed to support an expanding group of users. As these applications have grown, so has the operating system. Recent versions of the operating system, as well as the applications, require more and more memory. Some hardware, such as the AT and members of the PS/2 family, are capable of holding up to 16MB of memory, but DOS can use the memory above 1MB only in a limited fashion, such as through a virtual disk. DOS applications cannot run in memory above 640KB. Applications cannot address memory above 1MB, and the address space between 640KB and 1MB is reserved for hardware I/O routines.

DOS has been the single-tasking operating system of choice for the IBM Personal Computer family. It continues to be the primary 8086/8088-based single-tasking operating system and provides an interim solution to buyers of 80286/80386 processors. It is an entry-level operating system for each processor of the Intel family.

OS/2 Standard Edition

In addition to DOS 3.3 and the PS/2 family of computers announced on April 2, 1987, IBM announced OS/2 Version 1.0. This new operating system was to be available in the first quarter of 1988. The first customer shipment was in December 1987.

Several books have chapters devoted to explaining why there is a need for OS/2. The main reason is that application developers and users were requesting new functions and enhancements to existing functions. It was impossible to implement many of these in DOS and extremely difficult to implement others. The desire to meet these needs was the driving force behind the decision to develop OS/2.

Features of OS/2 Version 1.0

OS/2 Standard Edition Version 1.0 (OS/2 1.0) broke barriers imposed on DOS by the Intel 8086 architecture and took advantage of features in the Intel 80286 processor:

- Memory addressability up to 16MB
- Multiple concurrent applications
- Multitasking
- Protection among applications
- Protection between applications and the operating system

Because of the multiple concurrent applications and multitasking, OS/2 1.0 had other features:

- Interprocess communication (IPC)
- Dynamic linking
- Demand loading of code and data
- Scheduling
- Memory management

Disk management had to have a new look. Support for code page switching was required, because National Language Support (NLS) had been a part of the operating system design.

With a new operating system that had many more functions and handled multiple concurrent applications, the old method of obtaining system services through interrupts would not work. In OS/2 1.0, system services were obtained through calls to application programming interfaces (APIs). This provided a cleaner support structure for high-level languages and allowed the implementation of the system service to change without affecting the application.

The new operating system had many new functions and took advantage of the hardware, but the large number of established DOS applications had to be considered. OS/2 1.0 included a compatibility box or DOS box that would support most DOS applications.

OS/2 1.0 placed new emphasis on ease of use with its menu-driven structure and online help.

OS/2 1.0 was the first IBM Personal Computer operating system to have service and limited warranty support.

Important goals in the design of OS/2 1.0 included

- Maximum flexibility, providing services that most applications require and allowing for services to be added to the system. System interfaces can be extended by adding new function names to the system library. Subsystems can be added through user-defined dynamic link libraries.
- Stable environment, meaning that each program must see a uniform environment each time it runs, even though the system environment is likely to be different each time.
- Localization of errors, meaning that the system must be able to isolate errors, inform the appropriate application, and notify the user properly.
- Software tools approach, in that the system must be modular and each function must be a tool usable by an application.

The following list compares the features of DOS and OS/2.

	DOS	OS/2
Processor Architecture Supported	8086	80286
Maximum Memory Size	640KB	16MB
Number of Tasks	1	Multiple
Number of Concurrent Applications	1	Many
System Interfaces	Fixed	User-extendable
Number of Users	1	1
Batch Files	.BAT	.CMD

Operating Modes

The Intel 80286 can run in either “real mode” or “protect mode.” However, it cannot switch back and forth easily. When the processor is initialized, it is placed in real mode. Switching from real mode to protect mode can be done with a command, but there is no command to switch back to real mode. The developers of OS/2 had to do some special programming to accomplish the switching. Such switching is necessary because OS/2 supports two operating modes: DOS (real) mode and OS/2 (protect) mode.

DOS Mode

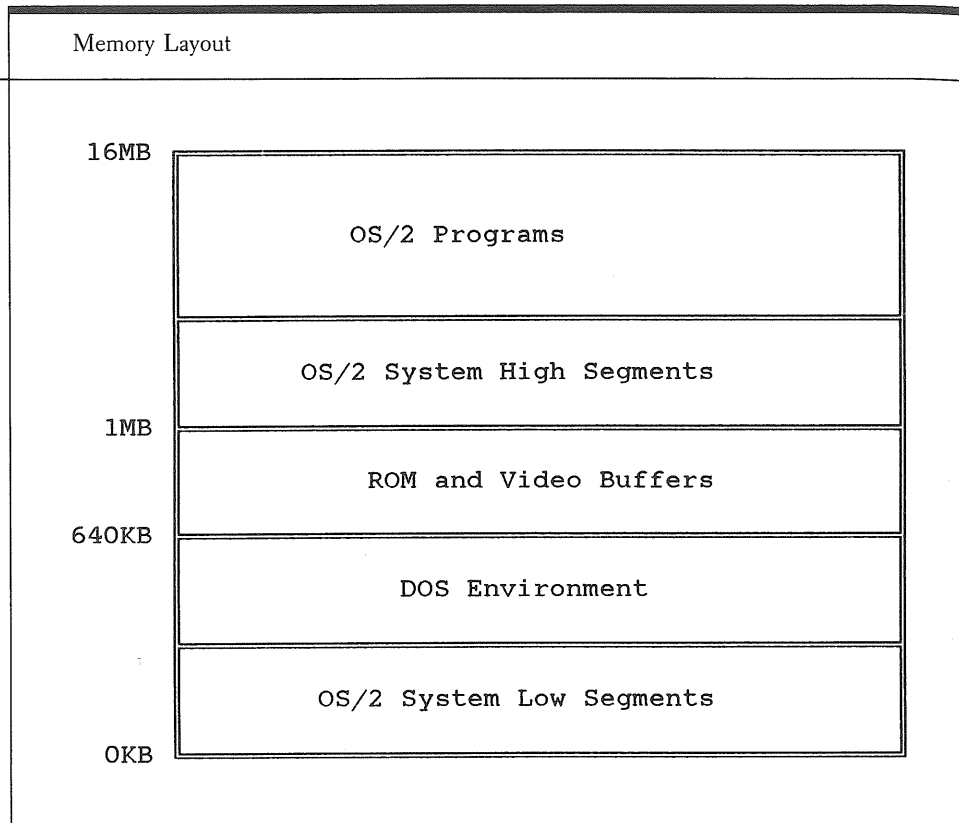
The DOS mode, referred to as the DOS box or compatibility box, is an environment that is similar to IBM DOS Version 3.3. COMMAND.COM is the command processor. Addressing is limited to 640KB. Programs can execute one at a time and only in the foreground.

OS/2 Mode

OS/2 mode is the multitasking environment that enables you to run more than one program written for OS/2 at a time. CMD.EXE is the command processor. Multiple sessions can be started, each with its own command processor, to run programs or process commands. Each session runs independently of any other OS/2 session. Each application can address up to 16MB of system storage. This is made possible through the technique of “storage overcommitment.” Each application has a virtual address space that is mapped to the physical memory by the memory management function of the operating system.

Figure 2.1 shows how memory is used in the OS/2 system.

Figure 2.1



Types of Applications

Applications that run under OS/2 1.0 can be divided into three types:

- OS/2
- DOS
- Family application programming interface (FAPI)

OS/2 Applications

OS/2 (protect mode) applications are written specifically for the protect mode of the operating system. They take advantage of the 80286 operations and use the new OS/2 API mechanism.

DOS Applications

DOS applications are those existing applications written for DOS 3.3 and previous DOS releases, as well as new applications that use only the facilities and capabilities of DOS. Most existing DOS applications, which are timing independent, can be run in the DOS box one at a time. Communications programs, network-dependent programs, hardware-specific programs, and interrupt-driven programs are timing dependent and cannot be run in the DOS box.

FAPI Applications

A program can be written to the family application programming interface to make it portable from OS/2 to DOS. Programs written using these special conventions can run in three environments: OS/2 protect mode, OS/2 DOS mode, and DOS 3.3. FAPI applications cannot use the new features of OS/2, such as memory addressability above 640KB and multitasking, but the capability to use a single source to create a program that will run in all three environments has its advantages and provides a migration mechanism.

Protection

An OS/2 application cannot inadvertently modify another application or the operating system. However, there is no memory protection in the DOS box. An errant application can destroy another application or the operating system in the address space assigned to the DOS box. Applications running above 1MB and the portions of the operating system above 1MB are protected from accidental or willful modification by any application in the system.

Multitasking

OS/2 allows the user to run multiple concurrent applications. Most applications appear as if they have the entire system to themselves, much as an application in DOS would do.

The idea of sharing resources among tasks is called *multitasking*. There are two basic types of multitasking: serial and parallel. In *serial multitasking*, one task at a time runs, but you can switch between tasks. With *parallel multitasking*, the CPU can run more than one task at a time.

Strictly speaking, OS/2 is a serial multitasking system, because there is only one CPU and that CPU can run only one task at a time. Each task receives its time on the CPU in a serial fashion.

However, the operating system can take the CPU away from any task at any time if a task with higher priority wants the CPU. Tasks with equal priority are given equal slices of time on the processor. With the preemption of tasks and time slices turning over many times a second, it appears that the system is indeed running several tasks at one time. The time-sliced, priority-based preemptive scheduling allows OS/2 to provide full multitasking.

Memory Management

The memory management scheme is based on special hardware facilities that are built into the Intel 80286 and 80386 processors. The memory is virtualized, so that each application appears to use all the memory. The total memory used by all applications running on the system at any time can exceed the physical memory of the hardware. In fact, a single application might use memory greater than the amount of physical memory. This capability is called *memory overcommitment*.

Memory overcommitment is handled by the technique of *segment swapping*. When a program requires more memory to be allocated for data or a program segment to be executed is not currently in physical memory, the memory management function takes the least recently used memory segment and writes it out to disk, making the memory available to satisfy the request. This implies that the memory management facility has much more to do than just keep the programs out of each other's way. It must keep track of each memory segment in physical memory. It must know who currently owns each segment and whether it is needed any longer.

Dynamic Link Libraries and Subsystems

OS/2 provides a *dynamic linking* facility. In the old technique of *static linking*, external functions referenced by a program are resolved at link time by copying the executable code for the external function into the executable module for the program. This makes the executable module for the program larger and requires the program to be relinked if the external function implementation changes.

With dynamic linking, the external references are resolved by placing special dynamic link records in the program's executable module. These records point to the dynamic link libraries where the function can be found. The executable code for the external function does not become a part of the executable module of the program. The external function is then loaded into memory, either at the time the executable module of the program is loaded into memory or when the program is running and calls the function. These functions are known as *load time dynamic linking* and *run time dynamic linking*, respectively.

Dynamic linking has several advantages. Program-executable modules are smaller. With multiple applications running concurrently, only one copy of commonly used external functions needs to be loaded into memory. The implementation of a dynamic link library function can be changed without any application having to relink.

Dynamic link libraries are used to provide subsystems for the programmer. Keyboard functions, video functions, and mouse functions are available through dynamic link libraries. These subsystems can be easily expanded because of the dynamic link library concept. Application developers can use this technique to provide their own customized subsystems.

Interprocess Communication

With multiple processes running in the system, the need arose to communicate among those processes. Processes may communicate using pipes, semaphores, queues, signals, and shared memory. Each method has advantages in different situations. The ability to communicate among processes allows the programmer to separate an application into tasks that perform specialized functions and yet appear as an integrated, logical program to the user.

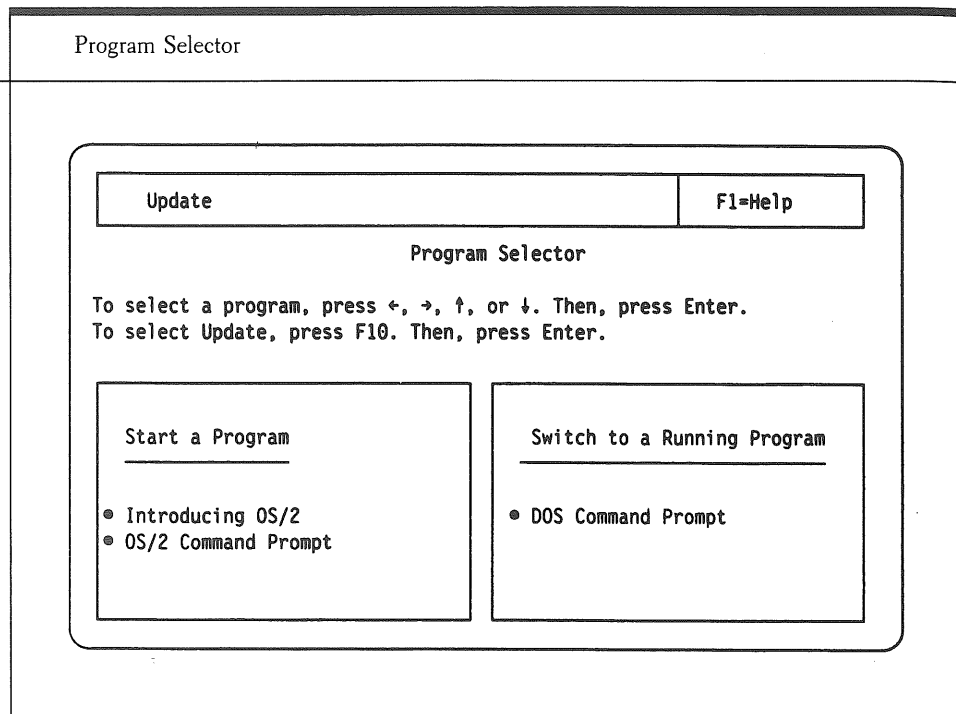
Program Selector

The user interface in OS/2 1.0 is character oriented. The video (VIO), keyboard (KBD), and mouse (MOU) subsystems provide the interface mechanisms. The VIO subsystem in OS/2 1.0 has no graphical services. Figure 2.2 shows the Program Selector, which is the main menu in OS/2 1.0. From this menu you can add titles of installed programs, start programs, switch between running programs, add or change program information on the Program Selector, and access the OS/2 command prompt or the DOS command prompt to install programs.

Device Drivers

Device drivers are used to pass information between the operating system core (called the *kernel*) and input or output devices. Typically, an application makes a call to the operating system that results in a function in the kernel being called. The kernel function calls the appropriate device driver. The device driver interacts with the physical device to perform the requested function and returns control to the kernel. The kernel then passes control back to the calling application.

Figure 2.2



DOS device drivers are usually synchronous and not interrupt-driven. They can be used only in DOS mode. OS/2 device drivers can be used in either DOS mode or OS/2 mode. OS/2 1.0 contains many device drivers, but the application developer or hardware developer can write device drivers to support additional devices and easily incorporate them into the system.

OS/2 Standard Edition Version 1.1

The intent to develop and market OS/2 Standard Edition Version 1.1 was announced on April 2, 1987, in the same announcement that introduced OS/2 Standard Edition Version 1.0. An availability date was announced in the fourth quarter of 1987. OS/2 1.1 was available on October 31, 1988. OS/2 1.1 is composed of OS/2 1.0, with enhancements and a major new component—the Presentation Manager.

This section briefly discusses the important enhancements to OS/2 1.0, which will be referred to as the *kernel*, as well as the features of the Presentation Manager.

Kernel Enhancements

The most important enhancements to the kernel in Version 1.1 included new features in the kernel device drivers, support for larger partitions on hard files, and a full-screen editor as a part of the operating system.

Previously, device drivers were restricted to one code segment and one data segment. An enhancement in OS/2 1.1 enables device drivers to have multiple code segments and multiple data segments. Device drivers are required to be loaded below the 640KB line. Because the device drivers have become more numerous and larger, the space available in the DOS environment for program execution is reduced. With the capability of multiple segments, the small interrupt handler and strategy routine can be loaded below 640KB, with the larger work routines above the 1MB line. This relieves some of the strain on the DOS environment, because the device drivers now use less low memory. This permits device drivers to perform more function without concern for the low memory restriction.

Another enhancement to kernel device drivers was the capability to make function calls among device drivers. This gave kernel device drivers a way to communicate with each other.

The PS/2 family of computers can accommodate 314MB fixed disks on some systems. However, until OS/2 1.1, the largest partition that could be defined was 32MB. This meant that several partitions had to be defined for a large fixed disk. In OS/2 1.1, this restriction was removed.

Presentation Manager

OS/2 1.0 could run concurrent applications in OS/2 mode. The system defined 16 sessions, 13 in which you could run programs. You could run applications in 12 OS/2 protect-mode, full-screen sessions and in 1 DOS mode full-screen session. The system used the 3 additional OS/2 protect-mode, full-screen sessions: 1 for hard error handling, 1 for VIO popups, and 1 for the Program Selector. The Program Selector allowed the user to start programs and to switch to running programs.

OS/2 1.1 uses the 16 sessions similarly to the way OS/2 1.0 used them. The main difference is that the Presentation Manager replaces the Program Selector. In addition to being able to start programs and to switch to running programs, the Presentation Manager runs extended sessions within its session. You can run up to 16 VIO-windowed extended sessions and up to 14 Presentation Manager extended sessions. If you disable the spooler, you gain an additional Presentation Manager extended session. A sixteenth Presentation Manager extended session is used for the Desktop Manager system application, which cannot be disabled. The Task List control facility also runs in a special

session within the Presentation Manager session. A *VIO-windowed application* is one that may be run in a window, but which is not aware that it is running in a window. A Presentation Manager application is one that is window-aware and can take full advantage of the functions offered by the Presentation Manager.

There are two types of Presentation Manager applications: graphics and Advanced Video I/O (AVIO). Graphics programs use a graphical interface to interact with the user. AVIO programs use an alphanumeric interface to interact with the user.

Windows

The Presentation Manager is a graphical user interface in a windowed environment. The use of windows allows concurrent access to applications and system utilities through the mouse and keyboard. With the Presentation Manager, not only can you have multiple concurrent applications running in the system but if they are running in the Presentation Manager session, they can be viewed simultaneously. The windows appear on top of each other like a messy desktop. Figure 2.3 demonstrates this window overlap display. The user can control the size and position of each window. There is a clipboard facility that allows the user to move or copy data from one Presentation Manager window to another if the applications running in those windows have provided that facility. The user can switch rapidly between the windows.

Presentation Manager applications can support one or more windows in a hierarchical relationship. Typically, a program defines a main window to represent the application and uses other windows on top of the main window to get input from the user and to display output. Windows can have menus and submenus that provide options to the user. Help information can be tied to windows to provide online assistance.

Graphics

The Presentation Manager provides advanced graphics support with integrated graphics functions and graphics development tools. Graphics functions include graphics primitives, attributes, and processing, including transformation and clipping, correlation, and bounds. Both retained and nonretained graphics are supported. There is a high-level interface for creating device-independent graphics.

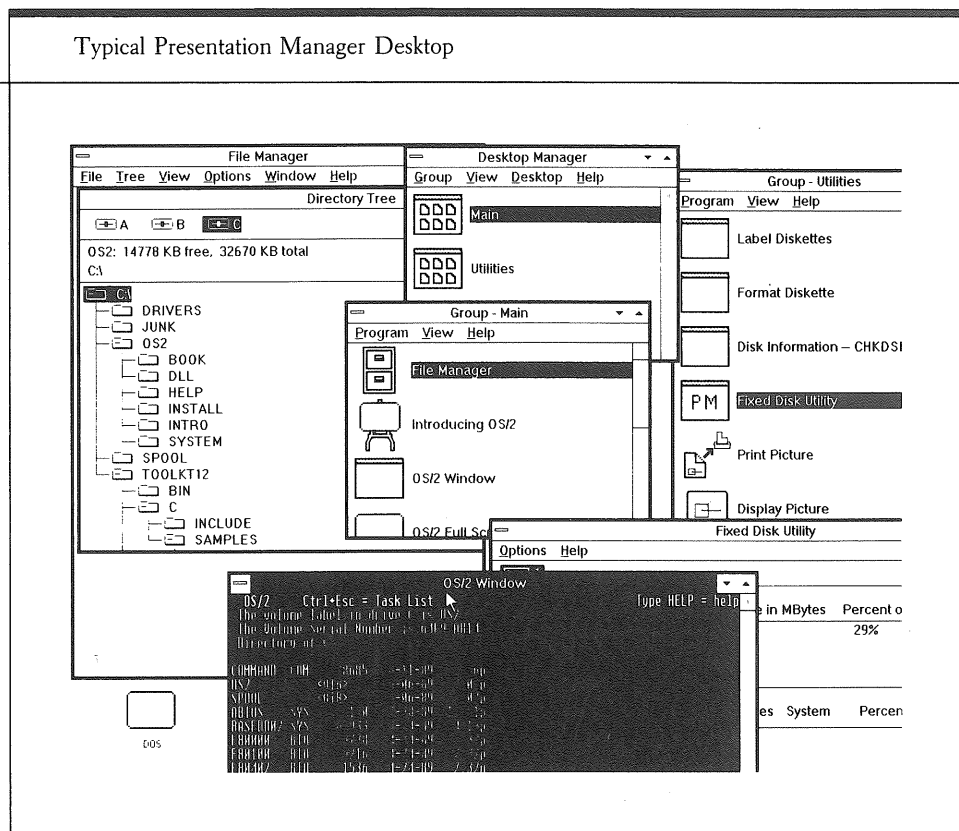
The use of the Presentation Manager's graphics functions makes it easier to produce more creative applications. A real estate agent could use this type of application to display the floor plan of a house, the home site plan, and other features of interest. A travel agent could display the layout of the airplane with available seats in one color and reserved seats in another color. A secretary writing a letter for her manager could use her word processor to retrieve graphics generated by another program and include the graphics in the letter.

User Interface

The user interface generally conforms to the Common User Access (CUA)[™] and Common Programming Interface (CPI)[™] of IBM's Systems Application Architecture

Figure 2.3

Typical Presentation Manager Desktop



(SAA)TM. There are easy-to-use system functions for system control, file maintenance, and file editing. Menus are used to prompt and guide the user through system facilities. Icons are used to reduce screen clutter and still provide ready access to programs and system functions. The system font is a proportional font that is easier to read than a fixed font. Various fixed and proportional fonts are available for the programmer to use in applications.

Pop-up windows provide messages to the user. Action bars, pull-down menus, and scroll bars are used in windows to help you to interact with a program. Specialized pointers and selection icons help to identify processing and functions that are available. The novice user can perform functions with a simple, slow, guided process, whereas the expert user has a fast path to system functions.

The file system is presented in a graphical tree format. Files and directories may be copied, deleted, renamed, and moved easily with the directory structure presented in this pictorial manner.

Device Drivers

The Presentation Manager device drivers translate device output requests to the appropriate OS/2 device driver. The device drivers support all-points-addressable (APA) devices.

Toolkit

The toolkit contains tools to support the graphics and windowing functions. Editors are provided to create and modify fonts, bitmaps and icons, and dialogs. A resource compiler is used to combine the resources built with the editors into the program executables. INCLUDE files provide definitions of structures, variable types, and function calls.

OS/2 Standard Edition Version 1.2

Presentation Manager enhancements and significant new features were added when OS/2 Standard Edition Version 1.2 was announced on May 16, 1989, with availability on September 29, 1989. Announced at the same time was the IBM OS/2 Programming Tools and Information Version 1.2 (also known as the Toolkit).

Major features include

- enhancements to the Presentation Manager facility
- High Performance File System (HPFS)
- Dialog Manager
- Information Presentation Facility (IPF)
- compatibility with DOS 4.0
- online documentation
- windowed System Editor
- IBM FORTRAN/2® Version 1.02 and IBM COBOL/2® language interfaces for Presentation Manager

Presentation Manager Enhancements

The Presentation Manager facility includes new functions and usability enhancements that help both the application developer and the user. Specific enhancements that aid the application developer include

- the ability to check for window visibility to avoid unnecessary window painting

- clipboard support in VIO windows
- new functions that manipulate program groups and programs within those groups
- availability of multiple font sizes for use within terminal emulation programs
- new picture utilities, such as PICPRINT, which prints metafiles and picture interchange format (.PIF) files; PICSHOW, which displays picture files; and PICICHG, which allows a .PIF file to be converted to a Presentation Manager metafile

Enhancements to the user interface include the Desktop Manager for adding and starting programs and the File Manager for displaying and manipulating files. Extensive use of icons in both the Desktop Manager and the File Manager makes it easier to perform desktop and file operations.

High Performance File System

The High Performance File System (HPFS) is a new, faster file system that supports large-disk media having as many as 16 partitions as large as two gigabytes each. Compatibility with the FAT file system is maintained at the API level. Names of files in an HPFS partition can be up to 256 characters long—longer than the old 8.3 FAT format.

Files in an HPFS partition can be accessed by programs running in the DOS compatibility box. However, programs running under DOS 3.3 or DOS 4.0 cannot access these files.

Dialog Manager

The Dialog Manager is a tool application developers employ to implement application dialogs that use Presentation Manager facilities to interact with the user. APIs are provided for dialog elements that deal with displaying dialogs, handling dialog variables, and creating and ending dialogs.

Dialog elements that are not a part of the application program logic (application panels, application command tables, application messages) are defined using the Dialog Tag Language (DTL). The Dialog Tag Language Compiler (DTLC) processes panels written in the DTL.

The Dialog Manager supports IBM Pascal/2[®] in addition to the other languages supported by OS/2.

Information Presentation Facility

The Information Presentation Facility (IPF) is used by the operating system for access to online help and documentation. In the same way, the application developer may use IPF functions to provide application-specific help. Help text is coded using a tag

language specific to the IPF and compiled using the Information Presentation Facility Compiler (IPFC).

Compatibility with DOS 4.0

The DOS compatibility box was enhanced to be compatible with a subset of DOS 4.0 function. Support is provided for display of more than 25 lines of text and for additional video modes available on the Personal System/2 displays.

Online Documentation

The Command Reference is available online, accessed from the Desktop Manager window. The Programming Reference, which comes with the Toolkit, may also be installed online. Use of these references is made easy by extensive search facilities, cross-referencing, and other user aids.

Windowed System Editor

The System Editor in OS/2 1.2 contains additional functions and is designed to run in a window. This enables the user to work with the System Editor and remain in the user shell with other applications visible.

IBM FORTRAN/2 Version 1.02 and IBM COBOL/2 Language Support

OS/2 Version 1.2 provides interfaces to the Presentation Manager for IBM FORTRAN/2 Version 1.02 and IBM COBOL/2.

Summary

OS/2 Standard Edition Version 1.0 was a giant step in personal computing operating systems. It provided significant new functions, enhanced ease of use, and a platform for future application growth. It broke the 640KB memory barrier. Multiple applications could be run concurrently while maintaining compatibility with DOS applications already written. A new approach to system services was taken, with calls to APIs replacing interrupts.

OS/2 Standard Edition Version 1.1 introduces a rich graphical interface in the Presentation Manager. Windows are used to view multiple applications simultaneously. There is a standardized user interface that handles user input from the keyboard and the mouse and displays output in both alphanumeric and graphics formats. An emphasis on ease-of-use functions makes OS/2 1.1 quicker for the user to learn and use.

OS/2 Standard Edition Version 1.2 continues the emphasis on ease-of-use functions. The Presentation Manager facility has a new look and feel that is designed to present

a more pleasant and easier-to-use interface to the user. New productivity tools for the application developer—the Dialog Manager and the Information Presentation Facility—are included. The new High Performance File System provides support for large media and is faster than the old FAT file system. The addition of online documentation and new language support in this version of OS/2 increases the benefits for both the user and the application developer.

SECTION TWO

**USING OS/2
VERSION 1.2**

This section describes the OS/2 1.2 operating system from the viewpoint of the user. Understanding how the user interacts with the system and programs written to run in that system is helpful when you are deciding how to design and implement a program. Chapter 3, "Operating Environments," covers the system as a whole and shows how a user views the sessions and different program types.

Chapter 4, "User Features in OS/2," looks at how the user works with the keyboard and the mouse to provide input. Interaction with windows is discussed in the section on using Windows. The section on the user shell deals with user interaction with the shell and the system facilities within the shell.

Operating Environments

The user is faced with three different environments under the OS/2 1.2 operating system:

- DOS compatibility box
- OS/2 full-screen mode
- Presentation Manager user shell

Only one of these environments, called a *session*, is visible on the display at a given time. The visible environment is in the foreground. The physical devices—such as the keyboard, the mouse, and the display screen—are logically associated with the session currently in the foreground. The environments that are not visible are in the background. The characteristics and capabilities of each environment are discussed in this chapter. You control which environment is in the foreground, and you can switch among environments using system “hot keys,” discussed in Chapter 4.

DOS Compatibility Box

In the DOS compatibility box, also called DOS mode, you see an environment equivalent to the DOS 4.0 operating system. The traditional command line prompt is visible and available for command entries.

When the DOS compatibility box is in the foreground, most applications written to run under DOS and applications written to the FAPI may be executed. When you switch to a different environment, placing the DOS compatibility box in the background, execution in the DOS compatibility box is suspended. This means that time-dependent applications, such as communications programs, should not be run in the DOS compatibility box unless you are careful not to switch out of DOS mode while that application is running.

The DOS compatibility box might be considered to be a separate computer, because there is no interaction with the programs in the other environments. Only one program at a time can execute in the DOS compatibility box. This program uses the entire display screen. The operating system provides no mouse support for programs in the DOS compatibility box. Mouse support must be provided by the application.

OS/2 Full-Screen Mode

OS/2 full-screen mode is equivalent to the OS/2 protect mode found in the OS/2 1.0 operating system.

You can execute system commands, OS/2 programs, and FAPI programs from the OS/2 full-screen command line prompt. Up to 12 OS/2 full-screen sessions can be active concurrently. As the name implies, these sessions use the full display screen. Therefore, only 1 OS/2 full-screen session is in the foreground and visible at a time. However, unlike the DOS mode, programs running in an OS/2 full-screen session continue to execute when the session is in the background.

Programs running in the OS/2 full-screen session can take advantage of OS/2 facilities that allow applications to interact and communicate. These programs can interact with programs in other OS/2 full-screen sessions, as well as with programs running in the User Shell session.

Programs written to run in OS/2 protect mode under the OS/2 1.0 operating system run in an OS/2 full-screen session under the OS/2 1.2 operating system. Programs written for DOS do not run in these sessions, however.

If a Presentation Manager program is started from an OS/2 full-screen command line prompt, the user shell recognizes that it is a Presentation Manager program and switches to the Presentation Manager user shell to execute the program. DOS mode programs cannot be started from an OS/2 full-screen session.

Presentation Manager User Shell

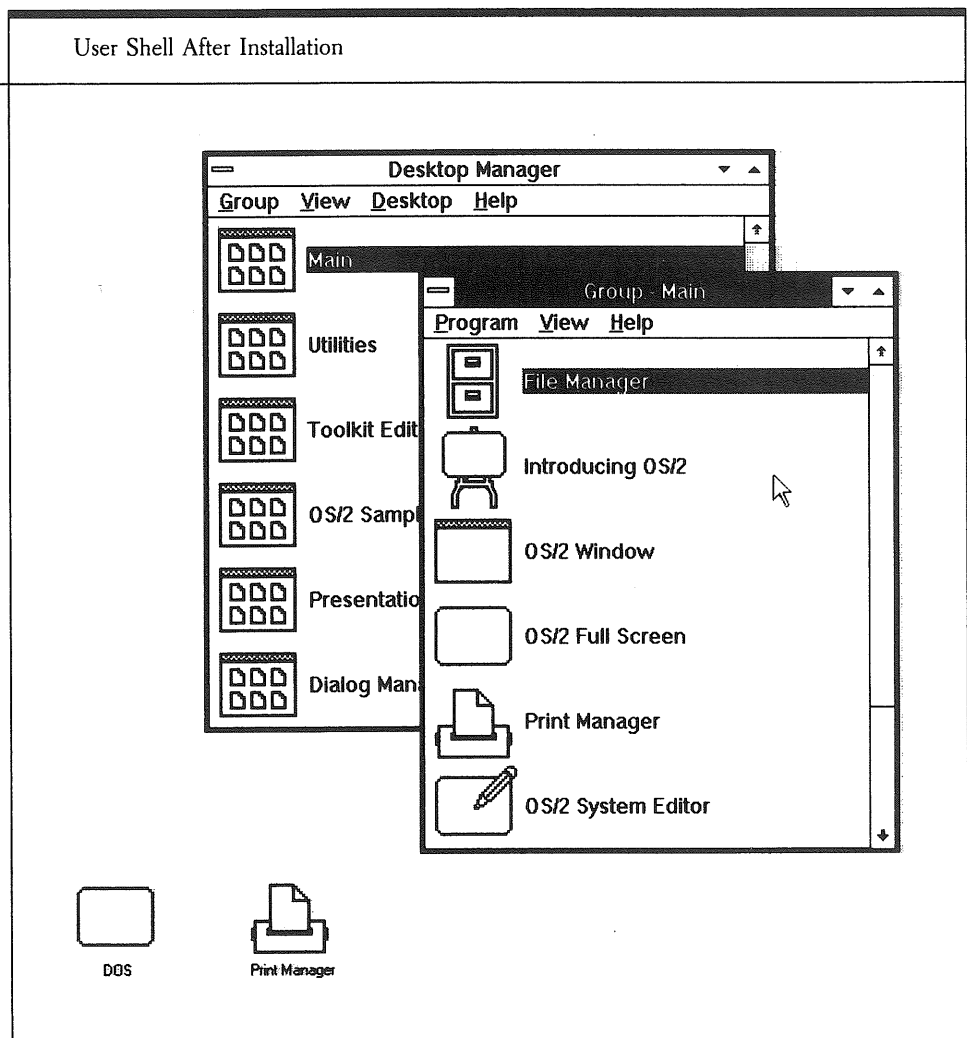
The Presentation Manager user shell is the newest and most important environment for the OS/2 operating system. The user shell is a special OS/2 full-screen session.

This session contains control facilities for the user and for the operating system. The details of these control facilities are discussed in Chapter 4. The user shell introduces the concepts of windows and extended sessions.

Windows

A *window* is a rectangular area of the display screen. In the user shell the entire display screen is referred to as the *desktop window*. Within the desktop window there can be many other windows. Refer to Figure 3.1 to view the user shell as it appears after the system is installed.

Figure 3.1



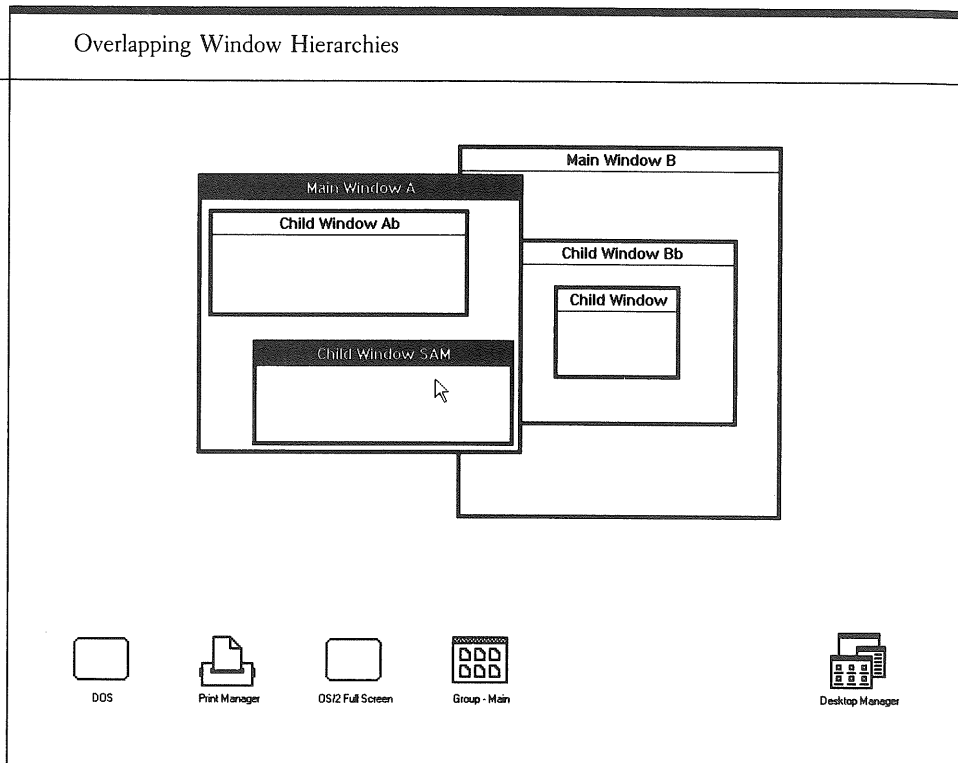
The Desktop Manager window is a main window for one of the user shell control facilities. A *main window* consists of many smaller windows and controls. These window parts are discussed in Chapter 4.

The symbols at the bottom of the desktop window are called *icons*. They represent applications whose windows have been minimized and sessions outside the user shell. The use of icons to represent applications within the user shell helps you to control the amount of clutter on the display.

A windowed application must have at least one main window. Separate asynchronous parts of the program running in the main window can have their own windows, called *child windows*. A child window lies on top of and is clipped to the borders of its parent window. Child windows may also have child windows, which are likewise limited to their borders. This layering of windows resembles pieces of paper on a messy desktop. Main windows are considered children of the desktop window.

A main window, its children, and all their descendants compose a *window hierarchy*. Window hierarchies representing one or more applications may overlap within the desktop window, as in Figure 3.2.

Figure 3.2

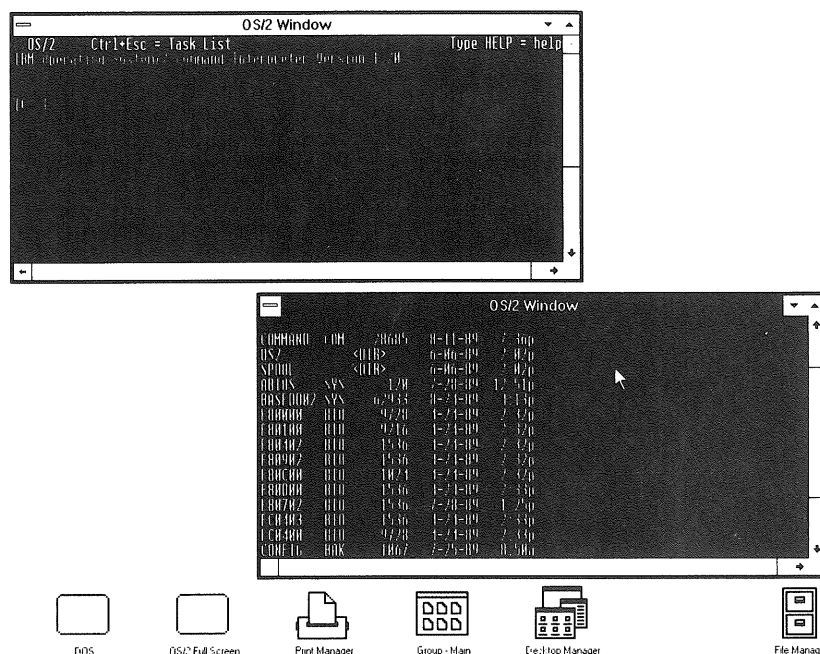


Children of the same parent are called *siblings*. Sibling windows have a display order, called *z-order*. Child windows higher in the z-order lie on top of children lower in the z-order.

There are two classes of extended sessions: VIO-windowable sessions and Presentation Manager sessions. The user shell and programs executing in the extended sessions continue to execute even when the user shell is in the background.

VIO-Windowable Sessions A VIO-windowable session is similar to an OS/2 full-screen session. An OS/2 windowed command line prompt serves the same purpose

VIO-Windowable Sessions



User Shell. The program may use one of two types of windows: advanced video input/output (AVIO) or graphics. A program running in an AVIO window uses alphanumerics to interact with the user. A program running in a graphics window uses graphics to communicate with the user. Each program running in a Presentation Manager session is aware that it is running in a window and presumably takes advantage of the facility. Up to 16 Presentation Manager sessions may be both running and visible on the display screen concurrently, along with whatever VIO-windowable sessions are running. Figure 3.4 shows an example of multiple Presentation Manager sessions and VIO-windowable sessions displayed concurrently.

Although there may be several programs running concurrently, only one program can be receiving keyboard input. The window associated with this program is said to have the focus.

Summary

The three environments in the OS/2 1.2 operating system provide compatibility with previous operating systems while introducing a new, easy-to-use interface. The DOS compatibility box enables you to run programs written for DOS. The OS/2 full-screen mode handles programs written for the OS/2 1.0 operating system. The Presentation Manager user shell introduces windows as the user interaction medium. The user shell is capable of executing and displaying up to 32 applications concurrently.

User Features in OS/2

Using the Mouse and Keyboard

The OS/2 operating system supports two physical devices the user can manipulate to provide input: the mouse and the keyboard. This chapter briefly discusses the terminology associated with each of these devices and explains how to perform the most common operating system tasks with OS/2.

The Mouse

You are not required to have a mouse to use the OS/2 operating system, but it sure makes life easier. The mouse can be used in two ways:

- It can be moved around on a flat surface.
- Its buttons can be clicked.

Most mice have two buttons, numbered 1 and 2. The left button, button 1, is usually the action button. (Buttons 1 and 2 may be swapped by the user through the control panel.) In all subsequent discussion, clicking on the mouse will refer to button 1 unless otherwise specified.

To *click* a mouse button, press the button and then release it. When instructions tell you to *double click*, press a mouse button twice, releasing it quickly.

Moving the mouse causes the mouse pointer on the display screen to move. An item can be selected by moving the mouse pointer to the item and clicking the active

mouse button. Another common function performed with the mouse is *dragging*. This is achieved by moving the mouse pointer onto an item, pressing the mouse button, and moving the mouse while holding the button down.

A new feature in OS/2 version 1.2 uses button 2 (the right button) as the “drag” button. Pointing to an object, clicking, and dragging with button 2 will move the object to where the pointer is pointing when you release button 2.

The Shift and Ctrl keys on the keyboard are used in combination with the mouse to achieve additional functions for extending a selection from a single item to a list of items.

The Keyboard

The mouse is an optional device in the OS/2 operating system; the keyboard is a required device. Some functions can only be accomplished using the keyboard. For example, switching out of the DOS compatibility box can only be accomplished with the “system hot keys” on the keyboard.

System Hot Keys There are two key combinations that are the main system hot keys: Alt-Esc and Ctrl-Esc. The Alt-Tab key combination has significance within the user shell.

Alt-Esc causes the next session to be displayed. What you see when you use Alt-Esc depends on what type of session is currently displayed and what type of session the next session is. Table 4.1 summarizes the scenarios.

The Ctrl-Esc key combination brings the Task List window to the foreground. If the current session is anything other than the user shell, pressing Ctrl-Esc switches the display to the user shell, and the Task List window is displayed as the topmost window. If the current session is in the user shell, the Task List window is made the topmost window. The purpose of the Task List is discussed later in the chapter.

Alt-Tab provides a function similar to the Alt-Esc key combination, except that it is used only within the user shell. When Alt-Tab is pressed, the next extended session within the user shell becomes the topmost window on the stack of windows on the display. Repeatedly pressing Alt-Tab cycles through each of the windows in the user shell.

Other Special Keys Other key combinations and single keys provide functions specific to the environment in which they are used. The use and meaning of some of these keys are discussed in the appropriate chapters in the remainder of this section.

Using Windows

Windows provide a natural, intuitive method for applications to communicate with the user. Normally you can reposition a window and change its size. Windows consist of

Table 4.1

Results of Using Alt-Esc in Various Sessions			
Current Session Type	Next Session Type		
	OS/2 Full Screen	DOS Compatibility Box	User Shell
OS/2 Full Screen	Entire Display Changes	Entire Display Changes	Entire Display Changes
DOS Compatibility Box	Entire Display Changes	Not Applicable	Entire Display Changes
User Shell	Entire Display Changes	Entire Display Changes	Extended session in user shell ¹

¹The window associated with the next session becomes the topmost window on the stack of windows on the display. This topmost window receives the keyboard focus.

parts and controls that are elementary windows themselves. Figure 4.1 shows the features a window might have. This chapter discusses how the user interacts with these window parts.

Selecting Menu Items

Windows typically have menus from which the user may select the option or command you want to use. For example, the System menu and pull-down menus from the Action Bar provide basic system options. There are various ways of activating these functions. The mouse or the keyboard may be used exclusively, or they may be used in combination. To simplify the discussion, the mouse and the keyboard actions are covered here as if each one were being used exclusively.

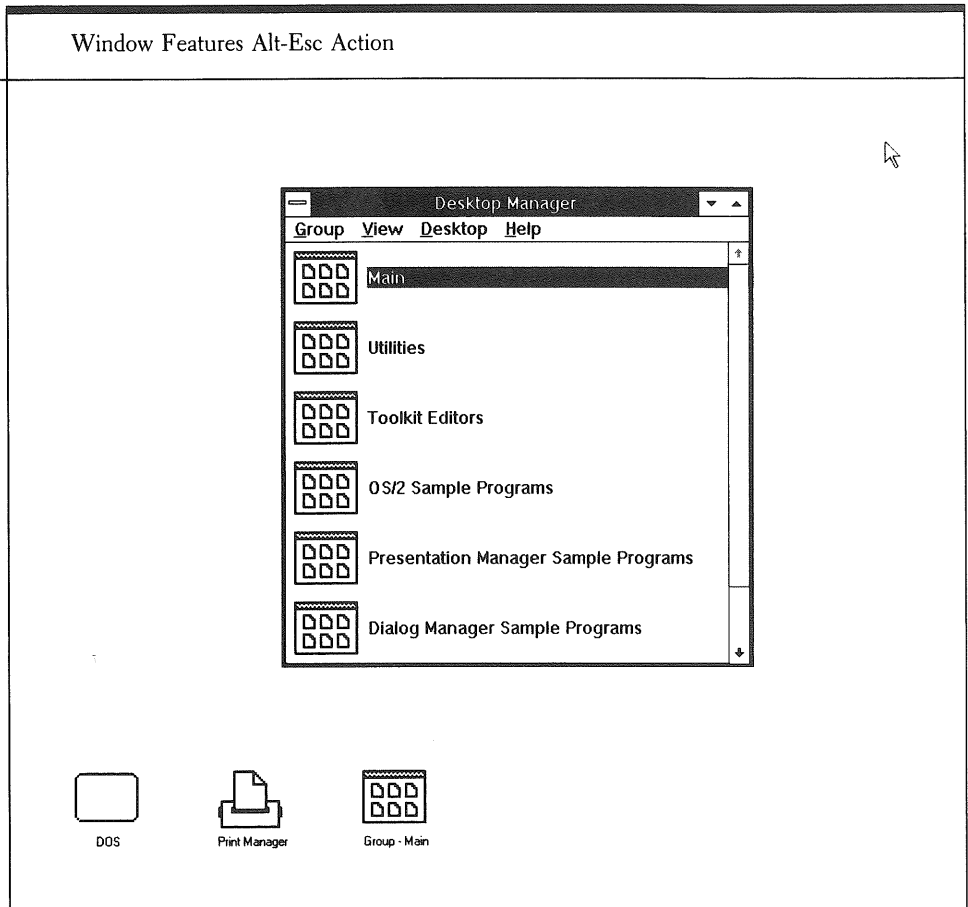
Using the mouse, you click on an icon, such as the System menu icon, or on an item, such as a word on the Action Bar. This displays a pull-down menu. A menu option is activated by moving the mouse pointer to it and clicking.

Menu access using the keyboard is a little more complicated. To activate an item on the System menu:

1. Select the System menu by pressing Shift-Esc or Alt-Spacebar.
2. Use the arrow keys to move the marker to the item to be selected.
3. Press Enter to select the item.

Figure 4.1

Window Features Alt-Esc Action



To get to an item on a pull-down menu from the Action Bar, follow these steps:

1. Press F10 or Alt to select the Action Bar.
2. Use the arrow keys to move to the desired choice on the Action Bar.
3. Press Enter to display the pull-down menu.
4. Use the arrow keys to move to the desired menu item.
5. Press Enter to select the menu item.

Two facilities provide shortcuts for menu selections: mnemonics and accelerators.

Mnemonics are shortcut entries (usually one character per menu option) that provide quick access if a menu is already displayed. An underscore under a character in the

menu item designates the mnemonic. Mnemonics are unique letters within a menu. If the menu is already displayed, you can press the key corresponding to a mnemonic and activate a menu item.

Accelerators are keys or key combinations displayed to the right of items on a menu that activate those menu items when you press the key(s) displayed. Accelerators for a window may be used whenever that window has the input focus. There is no need to have the pull-down menu displayed. Accelerator keys are unique throughout all the pull-down menus—including the System menu—within a window.

Moving Windows

Most windows can be moved using either the mouse or the keyboard. To move a window using the mouse:

1. Move the mouse pointer to the Title Bar of the window to be moved.
2. Press and hold the mouse button. (A ghost of the window borders appears.)
3. Drag the mouse until the ghost is in the position to which you wish to move the window.

Note: The Esc key can be pressed to cancel the move.

4. Release the mouse button. (The ghost disappears and the window is moved.)

Note: The system does not allow you to move the window off the top of the display such that the Title Bar totally disappears. This safety feature enables you to move the window with the mouse again later.

A window can also be moved by using the keyboard and the System menu:

1. Activate the Move option from the System menu. (A ghost of the window borders appears.)
2. Use the arrow keys to move the ghost around the display.

Note: Press the Esc key if you want to cancel the move.

3. Press the Enter key. (The ghost disappears and the window is moved.)

Note: Be careful when you move a window using the keyboard. The system does not prevent the Title Bar from moving completely off the top of the display, as it does with the mouse. If the Title Bar is moved off the display, the only way to move the window again is through the keyboard and the System menu.

Sizing Windows

You can change the size of a window using either the mouse or the keyboard. There are two extremes in size that are handled as special cases: minimized windows and

maximized windows. When a window is minimized, it is reduced to an icon and placed in the icon parking lot, usually at the bottom of the screen. When a window is maximized, it becomes as large as the application allows and it may fill the screen. When you make a window one of these extreme sizes, you can restore it to the size and position that it had prior to being placed in a minimized or maximized state.

Sizing, other than minimizing and maximizing, can be done with the mouse as follows:

- 1. Move the mouse pointer over the border of the window that you wish to enlarge or reduce. (The mouse pointer changes from a single-headed arrow to a double-headed arrow.)

Note: Windows can be sized horizontally (by placing the mouse pointer on a side border), vertically (by placing the mouse pointer on the top or bottom border), and diagonally (by placing the mouse pointer on a corner).

- 2. Press and hold the mouse button. (A ghost of the window borders appears.)
- 3. Drag the mouse in the direction that you want to size the window until the ghost borders, which have changed to indicate the new size, are in the position desired.

Note: The Esc key can be pressed to cancel the sizing operation.

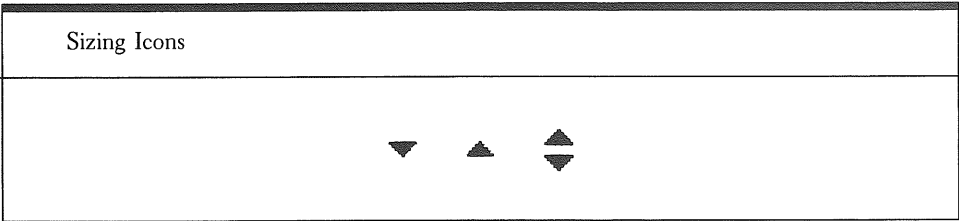
- 4. Release the mouse button. (The ghost borders disappear and the window is adjusted to its new size.)

The mouse can be used with the minimize icon, the maximize icon, the restore icon, and the minimized window icon to change the size of a window. Figure 4.2 shows examples of each of these icons.

A window in its normal (restored) state, when it is not minimized or maximized, may display minimize and maximize icons. If you click on the minimize icon, the window is minimized. If you click on the maximize icon, the window is maximized and the maximize icon is replaced by the restore icon.

A maximized window has two sizing icons: minimize and restore. If you click on the minimize icon, the window is minimized. If you click on the restore icon, the window is restored and the maximize icon replaces the restore icon.

Figure 4.2



When you double click on a minimized window icon, the window is restored. When you single click on a minimized window icon, the System menu is displayed.

You can also use the keyboard to display the System menu and perform sizing functions. The restore, minimize, and maximize functions can be activated from the System menu using the methods previously discussed. The result of activating these functions is the same as with the mouse.

To perform the intermediate sizing of a window using the keyboard and the System menu:

1. Activate the Size option from the System menu. (A ghost of the window borders appears.)
2. Use an arrow key to move the ghost border representing the window border that you want to adjust. The first arrow key pressed determines the border that is adjusted.

Note: The Esc key can be pressed to cancel the sizing operation.

3. Press the Enter key. (The ghost disappears and the window is adjusted to its new size.)

Note: Be careful when you size the top window border using the keyboard. The system does not prevent the Title Bar from moving completely off the top of the display, as it does with the mouse. If the Title Bar is moved off the display, the only way to size the window again is through the keyboard and the System menu.

Scrolling Data in Windows

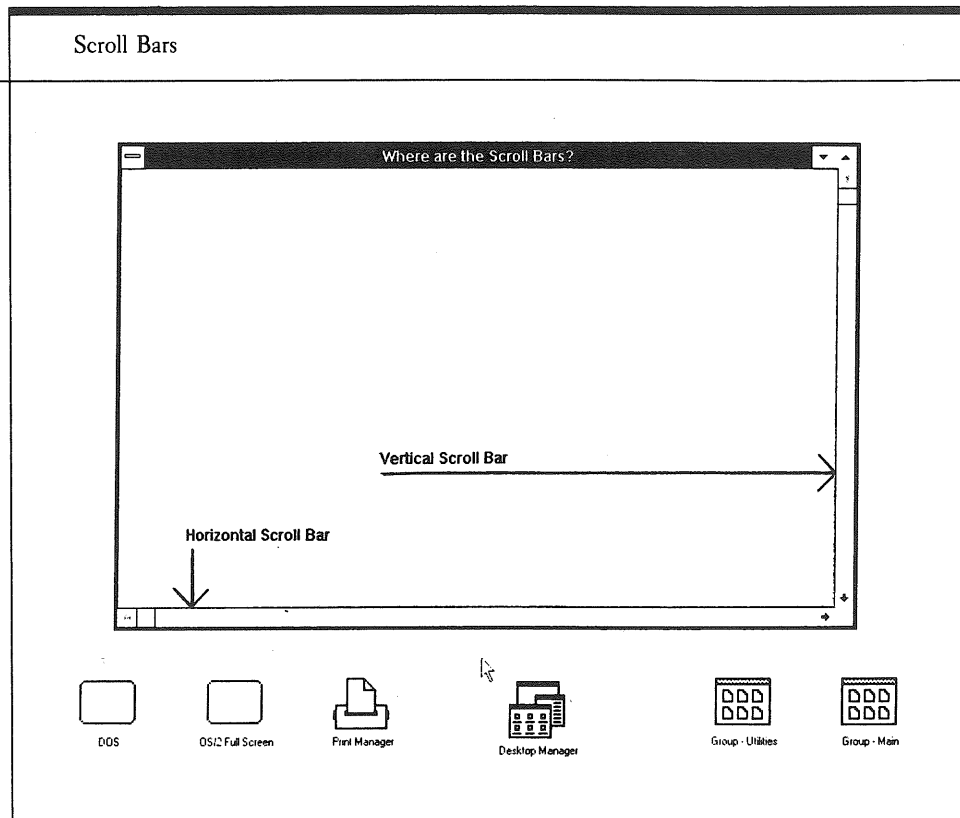
When a window is not large enough to display all the data available, you need a means to view the data not currently visible. OS/2 provides the scrolling facility. A window may have a vertical scroll bar, a horizontal scroll bar, or both. Figure 4.3 shows where the scroll bars are located in a standard window.

There are three ways to scroll using the mouse:

- Clicking on an arrow in the scroll bar causes the data to be scrolled one line vertically or one column horizontally.
- Clicking on the scroll bar between the slider and an arrow causes a page scroll.
- Dragging the slider causes scrolling relative to the distance the slider is dragged.

Scrolling can also be accomplished using the keyboard. Pressing the up or down arrow key causes scrolling by one line. Pressing the left or right arrow key causes scrolling by one column. Vertical page scrolling is accomplished by using the PgUp or F7 key to scroll up and the PgDn or F8 key to scroll down. Horizontal page scrolling

Figure 4.3



is done with key combinations: Shift-F7 or Ctrl-PgUp scrolls left one page and Shift-F8 or Ctrl-PgDn scrolls right one page.

Obtaining Online Help

Online help can be obtained for any window that includes Help on the Action Bar. Using the mouse to click on Help or pressing F1 from the keyboard causes a general help window for the window to be displayed. Specific (context-sensitive) help can be obtained for various parts of a window and menu items. Select an item with the keyboard or press and hold the mouse button when the mouse pointer is over the item for which help is desired, then press F1 to obtain specific help. (Release the mouse button if that technique was used.)

The help windows contain additional information you can access through the Help Index and the Keys Help options by selecting them with the mouse or keyboard.

Help is sometimes available when system error and warning messages are displayed in message boxes. Use the mouse or the Enter key to access this help.

Online help for system messages is also available at command line prompts by typing **help** followed by the message number.

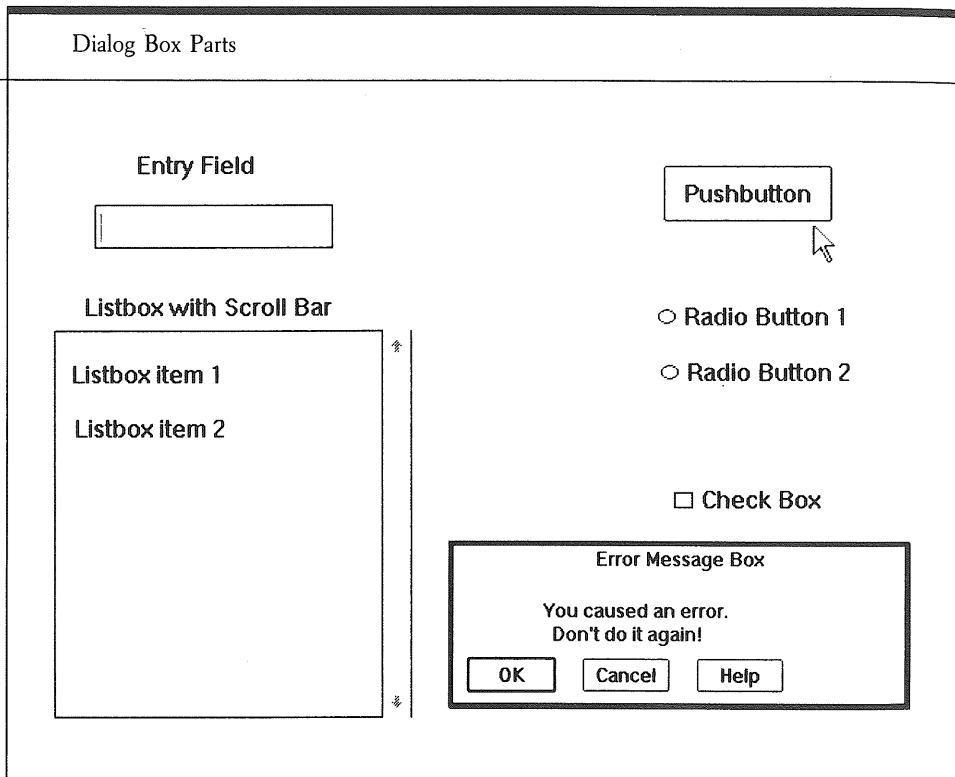
Using Dialog Boxes

Dialog boxes are special child windows that are designed to be displayed for a short time. They give you information and accept the information you type, just as the name dialog implies. There are different methods of communication used in a dialog box. Figure 4.4 shows a dialog box with some of these methods. Each one will be described briefly.

Entry Fields An *entry field* is a single horizontal line in which data may be displayed for editing and in which you can type input data from the keyboard.

Scroll Bars *Scroll bars* are used primarily to scroll through a list of displayed information.

Figure 4.4



Pushbuttons Text bordered by a rectangle is defined as a *pushbutton*. An action is performed immediately when the button is pushed (selected with the mouse or the Enter key).

Radio Buttons *Radio buttons* are shown as circles that are filled when you select them. They are used with a list where only one item can be selected.

Check Box *Check boxes* are indicated by a small box. A check mark appears in the box when the item is selected. Multiple selections may be made.

Listboxes A *listbox* is a rectangular area with a vertical list of text items, one or more of which is selectable. When such lists are large, you can scroll through them by holding the mouse button down on the up or down solid arrow. An item is highlighted when you select it with the mouse or keyboard.

Combo Boxes A *combo box* is a combination of a listbox and an entry field. Depending on the type, the entry field will display the currently selected listbox item or an entry the user has typed.

Multiline Edit Controls A *multiline edit control* is simply an entry field that can handle multiple lines.

Message Boxes *Message boxes* are used primarily for error or warning messages. They are usually displayed in the center of the screen. A message box must be responded to before the application can continue. In some cases, the message box must be taken care of before you can switch to anything else in the system.

Clipboard Functions

The Presentation Manager provides basic support for clipboard functions. Functions that may be available in a window include copy, cut, and paste. These functions work only within Presentation Manager windows. They are not available outside the user shell.

Although the basic support for these functions is in the user shell, an application must specifically provide support for these functions and make them available to users. This is usually done through an option on a pull-down menu.

With the copy function, you mark a portion of a window and copy it into the clipboard, leaving the window intact.

The cut function is similar to the copy function, except that the portion of the window that goes into the clipboard is removed from the window.

The paste function complements the other two functions. You use it to insert data from the clipboard into a window.

Even though these functions must be provided by an application, they are under the control of the user. No clipboard activity takes place unless the user requests it.

The User Shell

The Presentation Manager user shell provides five main areas of features that are the basis for controlling and customizing the operating system. These areas include a facility to install and start programs, a facility to control tasks that are running in the system, a facility to access and manipulate files, a facility to set user preferences and customize an installation, and a facility to control printed output.

Desktop Manager

The Desktop Manager facility is the repository for program installation and execution instructions. Figure 3.1 illustrates the Desktop Manager window and the Group - Main window as they appear after operating system installation.

Installed programs are placed in groups. Each group is an entry in the Desktop Manager window. Double clicking on a group name displays the group's window. There are three sets of functions, in addition to the Help functions, available through the Action Bar: Group, View, and Desktop.

Functions under the Group option include opening a group, adding a group, deleting a group, and renaming a group.

The View option enables you to choose whether you want icons to appear next to group names.

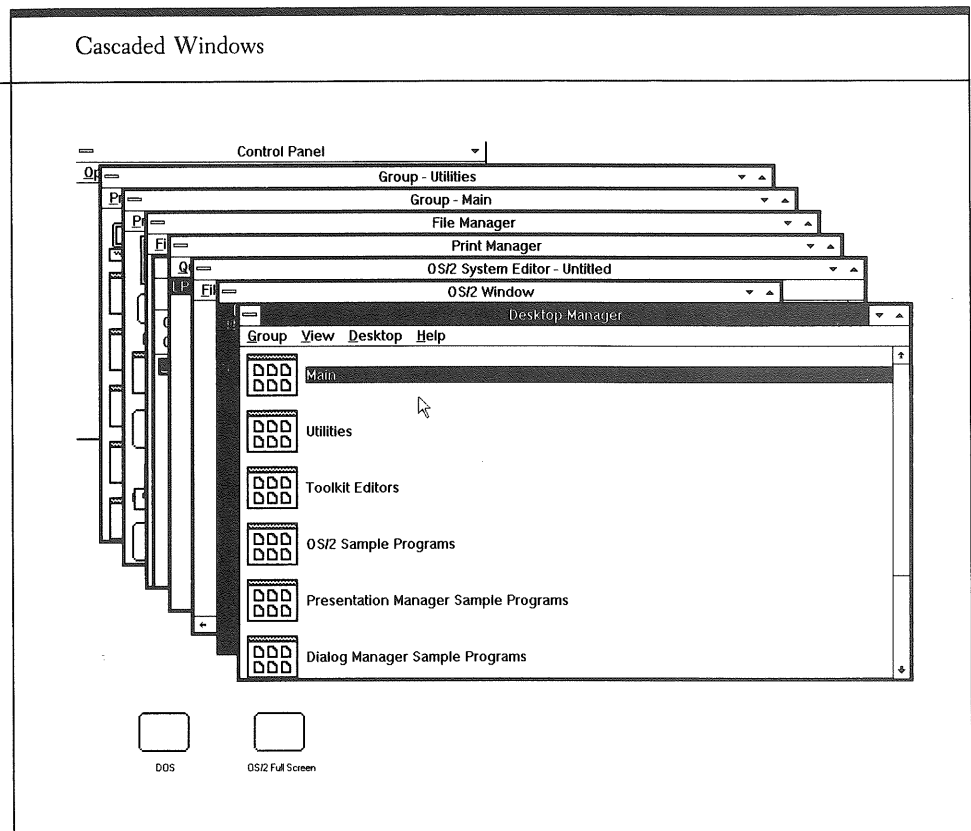
The Desktop option provides two classes of functions that affect the desktop. The top three functions are methods of arranging windows and icons on the desktop. Windows may be cascaded or tiled.

Windows are cascaded by placing the window lowest in the z-order in the upper left corner of the screen. Each successive window in the z-order is placed on top of the preceding window slightly lower and to the right, leaving a portion of the previous window showing on the left and on the top. This process continues until all windows are cascaded. If the number of windows is too great to be handled in one cascade, additional cascades are done starting to the right of the previous cascade. Figure 4.5 illustrates the effect of cascading.

When you choose to tile the windows, the display is divided such that all windows are visible. This means each window gets smaller as the total number of windows grows. Figure 4.6 shows the effect of tiling.

The bottom three functions under the Desktop option are concerned with closing windows and shutting down the system. You can choose to save the windows and icons

Figure 4.5



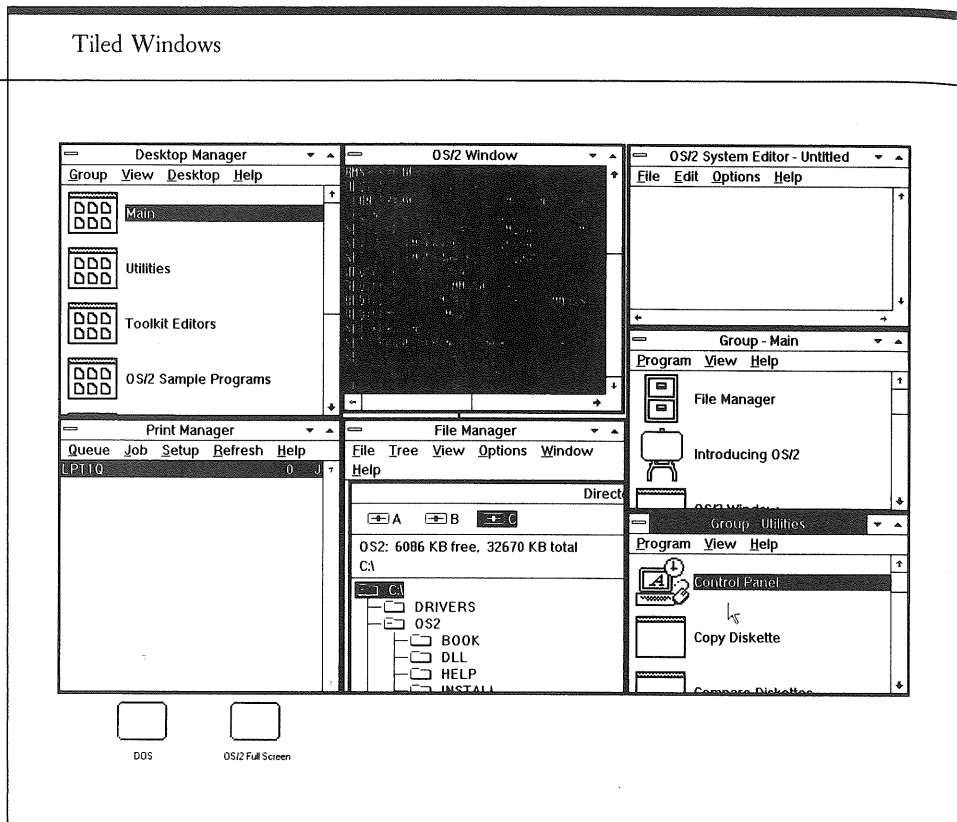
as they currently appear. The Presentation Manager programs that support this feature store this information so that the next time the programs are started, the saved appearance is used. The Close all choice ends all your running programs except the Desktop Manager, the Print Manager, and DOS. Shutdown is used to end all running programs and shut down the system. Shutdown should always be used to shut down the system before powering off the computer if you are using the High Performance File System.

The Program option on the Group window Action Bar provides facilities to start a program, add a program to the group, copy the program to another group, delete the program from the group, and change the information about the installed program properties.

The View option enables you to choose whether you want icons to appear next to program titles.

Programs may be started by double clicking on the program title with a mouse or by selecting the program title and pressing Enter on the keyboard.

Figure 4.6



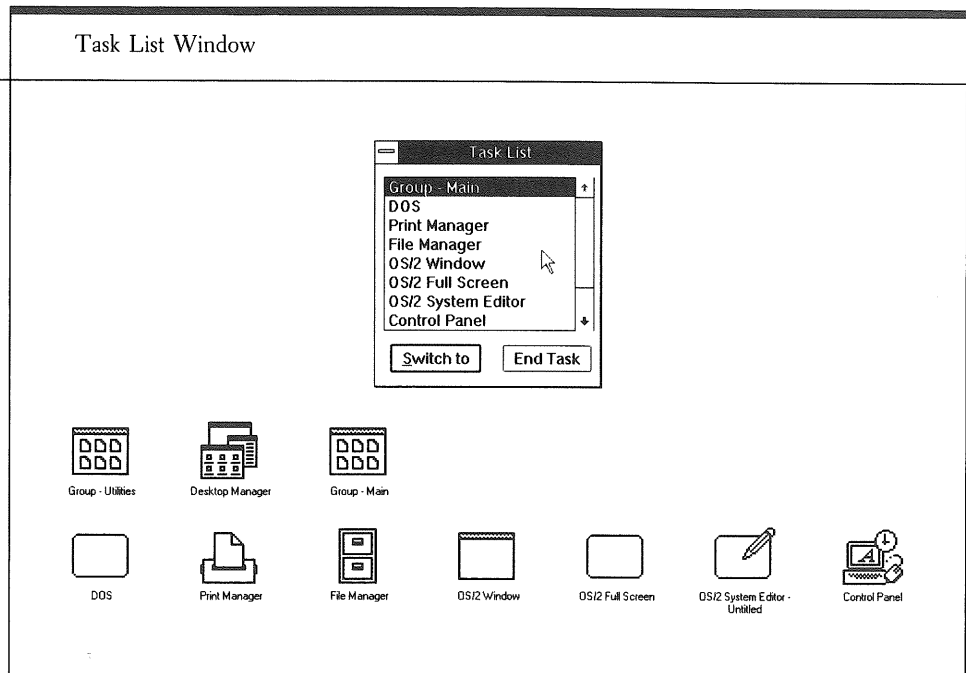
When a program is installed, the operating system recognizes whether it is a Presentation Manager program. If it is not a Presentation Manager program, you are asked for additional information to determine the type of session in which to run the program. You also have the option of telling the system that a program is a Presentation Manager program anyway. This is not advisable unless you know what you are doing. The results of running a non-Presentation Manager program as a Presentation Manager program are, at best, unpredictable. The program may cause a trap and terminate.

Task List

The Task List window displays a list of all the programs currently running in the system except those programs that specifically chose not to have an entry in the Task List. Figure 4.7 shows a typical Task List window.

The Task List Action Bar provides two options: Switch To and End Task. The End Task option enables you to terminate a running program. The other function

Figure 4.7



tells the user shell to switch to a running program. When this is selected, the session in which the program is running takes over the display.

File System

The File System in the OS/2 operating system provides a windowed, graphical method of displaying directory structures. The Directory Tree window shows the directory structure for a particular drive. Various options control the amount of data displayed and the format it assumes. Figure 4.8 shows typical file system displays.

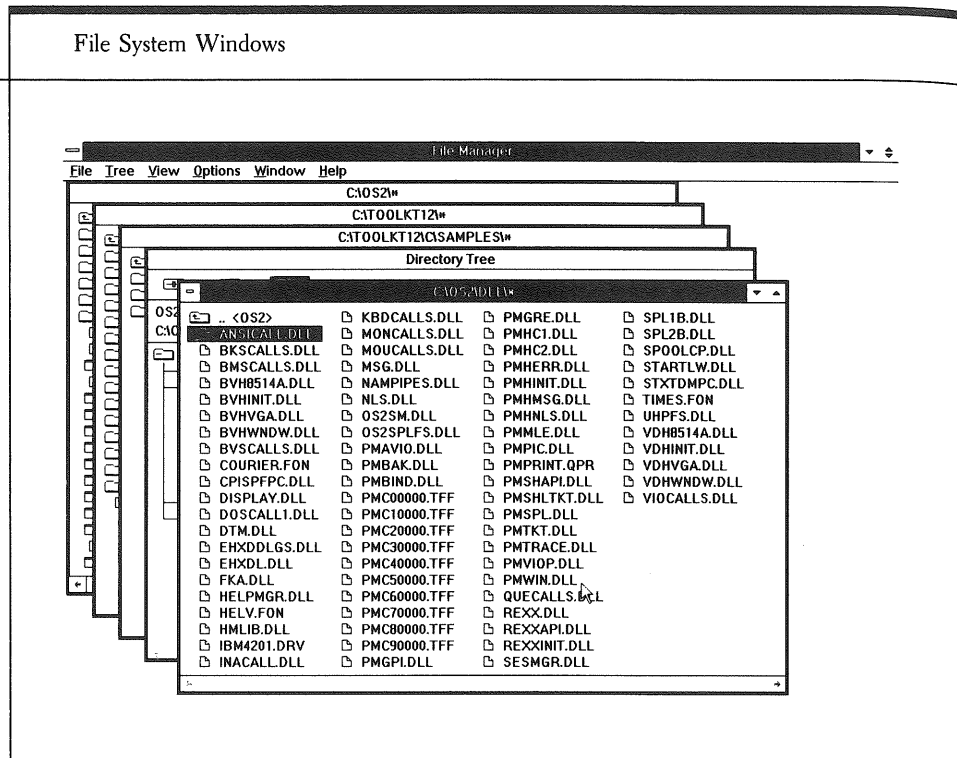
Note: The Refresh function on the pull-down menu from the Window option is a valuable tool when you use removable media. When you replace a diskette with a different one, the directory structure in the Directory Tree window is not updated unless you use the Refresh function.

With the directory structures displayed in windows and mouse support available, it is much easier to move and copy files and directories. There is a tremendous variety of features available in the File System facility.

Control Panel

The Control Panel facility is accessed from the Utilities Group. This facility contains functions you select to customize the system. Figure 4.9 shows the Control Panel as it is initially displayed.

Figure 4.8



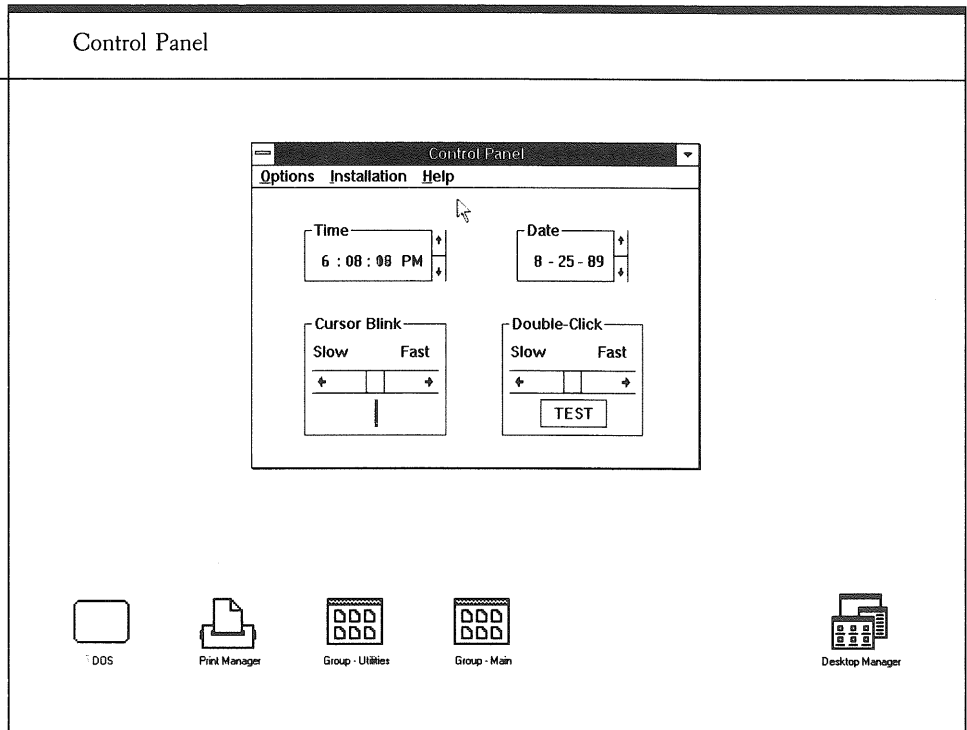
You can perform any of the following Control Panel functions:

- Set screen color preferences.
- Set the time and date.
- Define the rate of cursor blink.
- Define the speed of double clicks.
- Set mouse options.
- Set country information.
- Set up communications ports, printers, and spooler queues.
- Add and delete fonts, printer drivers, and queue processors.

Print Manager

The Print Manager is the feature you use to control the spooler and the print jobs sent to the spooler. Figure 4.10 shows the Print Manager window.

Figure 4.9



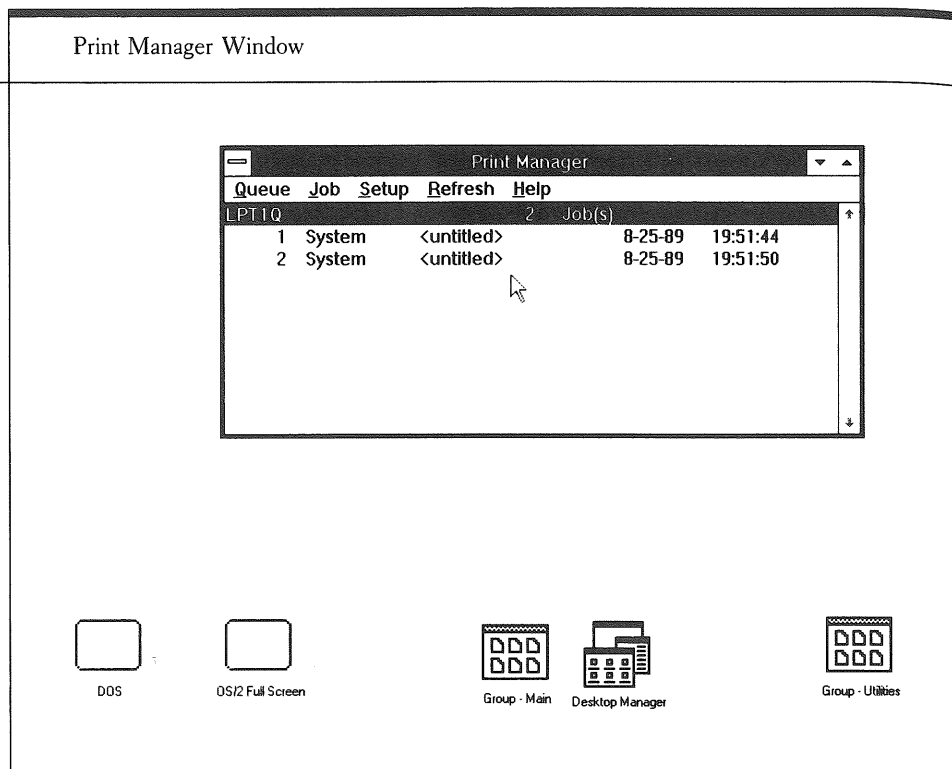
The options on the Action Bar access functions that control the spooler queues and individual print jobs. The print jobs currently in the queue are listed in the window, also.

Summary

The OS/2 operating system supports the mouse and the keyboard as methods for the user to interact with the system and applications. The mouse is well suited for a user-oriented, graphical interface such as that provided by the Presentation Manager facility. For those functions you cannot accomplish with a mouse and for users who prefer not to use a mouse, OS/2 provides support for keyboard interaction. Special keys are defined to accomplish various system functions.

Windows play an important part in the OS/2 operating system. Standard window parts and controls provided and supported by the Presentation Manager facility make it easier to write programs that are consistent in their interaction with users. You simply become familiar with a few basic terms and actions relative to windows. At that point,

Figure 4.10



interaction with any program conforming to the operating system guidelines is easy to learn and perform quickly and accurately.

The user shell provides facilities to control the tasks in the system and to customize the operating system. Facilities such as Desktop Manager and Task List deal with program installation and execution.

The File System facility has all the functions available in previous operating systems. In addition, it provides a graphical interface that enhances its usability.

The Control Panel enables you to set your preferences and install additional printers. The Print Manager helps you control printing and spooling.

SECTION THREE

PROGRAMMING CONCEPTS

This section describes the principles of programming in the OS/2 Standard Edition Version 1.20 environment. Although this book focuses on the OS/2 Presentation Manager system, to discuss all the programming aspects of this system in detail would require several thousand pages. In the interest of discussing the Presentation Manager system thoroughly, other aspects of the OS/2 system are addressed only as they relate to the Presentation Manager part of the operating system.

As an aid to understanding, we distinguish the core of the operating system, called the *base*, from the Presentation Manager system. The term *base system* describes the system facilities that handle detailed memory management, multithread programming, pipes, semaphores, and so on. The term *Presentation Manager system* refers here to all aspects of the Presentation Manager screen group: windowing, graphics, and the Presentation Manager shell.

TH

OS/2 Base System Features

The base portion of OS/2 Standard Edition Version 1.2 can be thought of as all the elements of Version 1.0. These functions form the core of the operating system. The base system handles memory management, interprocess communications, resource sharing, device drivers, basic screen and keyboard I/O, and the file system. It also includes the execution environment through which applications use the hardware to perform their functions. The fundamental portions of the base are described here to aid understanding of the Presentation Manager system.

The base system contains a rich set of functions that enable programs to access its capabilities. OS/2 programs use a high-level language Application Programming Interface (API) to manage resources and allow the operating system to control execution.

The Multitasking Model

The most prominent feature that distinguishes OS/2 from other PC operating systems is *multitasking*, a characteristic that allows many applications to run concurrently. Under DOS, there are many programs available that simulate this function, but OS/2 multitasking is implemented with functions native to the 80286 and 80386 processors. Applications run in a prioritized, preemptive, multitasking environment in which each application is protected from any ill behavior of others. Each thread in the system has a priority. The scheduler and dispatcher keep threads from monopolizing the system and coordinate execution among them. Each equal priority thread is allocated an amount

of processor time and is preempted when its time is up. In this fashion, OS/2 provides all applications a chance to execute.

Multitasking means that several tasks may run in a system at once with a single processor. This is what the OS/2 system enables computers to do. With only one 80286 or 80386 processor, several applications may run concurrently. Each of these applications may have several small units of work running concurrently within it. With all of these small units of work running in the system at once, the system must incorporate some kind of control and coordination. The 80286 and 80386 processors have a set of facilities and protection levels that keep all of this in order.

The highest level in the control hierarchy of the system is the *screen group*. Each screen group is a logical grouping of processes that is separate from every other screen group. Each one has its own set of logical resources that map into the physical ones, such as the screen or keyboard. The term *session*, which was introduced earlier, is sometimes used to refer to a screen group.

Each session represents a logical unit of system resources. Although there is only one physical device, each session has its own logical keyboard, mouse, and screen. The operating system arbitrates the connection of the logical devices with the physical devices. Each session can be thought of as its own "virtual machine."

One of the sessions is the user shell (also called the *Presentation Manager session*). Within the user shell there are two extended session classes: VIO-windowed and Presentation Manager.

A VIO-windowed session runs a subset of full-screen applications in the windowed environment without the applications being aware of the window. A maximum of 16 of these sessions can be active.

The Presentation Manager session runs full-function Presentation Manager programs. There are two types of Presentation Manager programs: graphics-based and alphanumeric-based. Both types of programs may use all the facilities of the Presentation Manager session and are fully aware that they are running in a window. Graphics-based programs use graphics and images to communicate with the user. Alphanumeric-based or Advanced Video I/O (AVIO) programs use textual data to communicate with users. You can use a maximum of 16 of these Presentation Manager sessions.

All programs in the user shell use the same screen and keyboard. Usually only one is designated as the focal point for console I/O at any point in time.

Next in the hierarchy is the process. A *process* is a unit of an application within a session. A process is established by the operating system as a control point for the ownership of resources. These resources include disk files, memory, threads, semaphores, and queues. For example, when a program is executed, the system creates a process to own everything associated with that program. All program resources—such as memory, files, and semaphores—are owned by a process. Some applications may create other processes within their own session so they may do separate logical units

of work. This enables each process to have its own set of resources and avoid resource conflicts.

For example, a program creates a process to run. This process has several data areas and threads that perform its processing. In the scope of the whole application, there may need to be several other threads and data areas that must be protected from accidental modification by this process. So, the application may create other processes to handle that work. Each process owns its resources, thus protecting the resources from each other. One process cannot touch the resources of another unless the resources are specifically programmed to be shared. This is another level of security by which an application may protect certain resources from itself.

The smallest unit of work within OS/2 is the thread. The *thread* is the basic unit of execution. It contains the information programs require in order to run. Each thread has its own stack, set of registers, and processor state relative to its instance of execution. This information is called the thread's *context* and is stored in an internal structure called the *Thread Control Block (TCB)*. Each thread in the system has its own TCB that contains all of the information it needs to run. The thread is what programs use to execute their instructions. There may be many threads running concurrently in the system.

To tie all this together, a session is started in which you execute a program. First, you enter the command to start the application. The system creates a process to control the program resources to be used during its execution. A thread is then created to execute the instructions in the program. This program may then execute code to start up other sessions, processes, or threads. They all execute concurrently with this one and, using the protection scheme discussed in the next section, are protected from each other. Sessions may not interfere with other sessions' programs, and processes may not interfere with the resources of other processes unless they are programmed specifically to share resources.

OS/2 Protection

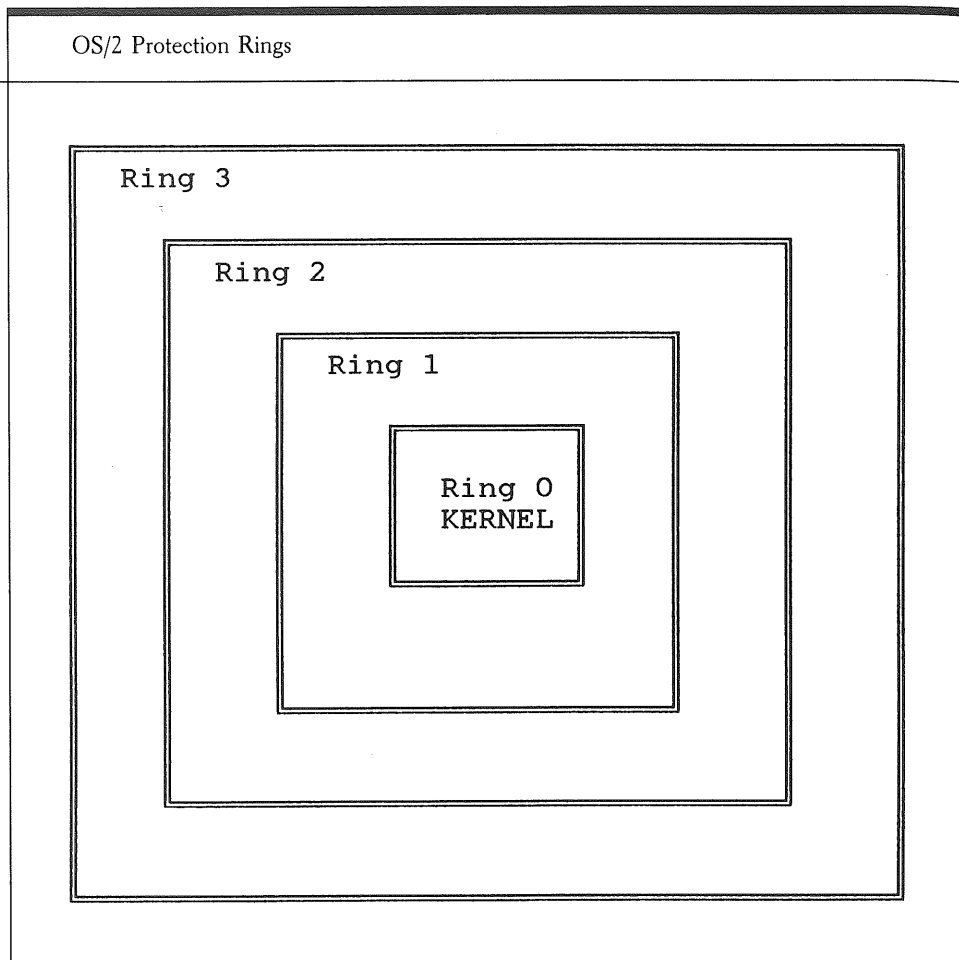
The protection scheme that is built into the 80286 and 80386 processors keeps applications honest. It is around this protection scheme that all the OS/2 subsystems are designed. It uses a "ring" protection method, which can be pictured as a series of concentric circles. The rings around the center represents the protection rings around the hardware. The bull's eye represents the kernel of the operating system and is known as ring 0. Three rings surround this kernel; they are numbered 1, 2, and 3. Each represents a privilege level, with the lowest level, ring 0, the most privileged, and ring 3 the least privileged. Code running at a particular ring may access data of code running at a lesser privileged level, but not the other way around. This is how the services at

more privileged levels operate on behalf of less privileged applications. Figure 5.1 shows this protection scheme.

Code running at ring 0 is the most privileged code in the system. It has to be, because the rest of the operating system uses it to communicate with the computer hardware.

The *kernel*, as the name implies, is the core of the operating system. The kernel runs at ring 0, executing the routines that provide all the services of the hardware to the application programs running in less privileged rings. This code directly controls hardware such as memory, processor registers, and the processor modes. This is the engine driving all of the hardware, serving everything else in the system. It contains code that manages thread switching, priority, memory, and the computer's bus.

Figure 5.1



Along with the kernel, device drivers run at ring 0. *Device drivers* are special programs that enable the kernel to talk to certain hardware devices. Examples of the devices are the mouse, keyboard, and the display. Device drivers are the translation and hardware management routines that provide the kernel access to the devices necessary for system function. Device drivers are loaded into memory at system initialization time and become extensions of the kernel.

The kernel uses device drivers by passing *packets* of information to them so the driver knows what is expected. Packets may contain instructions to the driver to return certain information about the device or to tell the device what to do, such as clear the screen or retrieve a sector of data from the disk. This is how the operating system communicates with its devices.

Another item included in the group of code running at ring 0 is the hard-error handler. This session is always running in the system for the sole purpose of monitoring for applications that generate severe errors. Severe errors include problems such as the disk drive not being ready, as well as any protection violations. Such errors occur when a program attempts to violate the protection rules in the system. They are known as General Protection Faults, segmentation violations, or "Trap D." When this type of error occurs, the hard-error handler informs you and terminates execution of the offending program.

Ring 1, which is immediately above the kernel in the protection model, is not used by OS/2 Standard Edition Version 1.2.

Segments of code that require an I/O privilege level (IOPL) run at ring 2. Enabling and disabling interrupts as well as direct I/O to the hardware are examples of IOPL. Some of the routines in the subsystems need this level of hardware access; that is why they run at ring 2. The code running at ring 2 has access to the code and data of applications and other subsystem routines running at ring 3. This facilitates the performance of requests by the subsystems on behalf of application programs.

Application programs and most of the subsystem routines run at ring 3, the lowest privilege level. This is done to protect applications from bad behavior of other programs. Programs executing at this level have access to only their own code and data segments or those of cooperating applications. Applications may share resources only as designed by the software authors. Otherwise, no application can access or change the code or data of another.

If a program requires system services, such as memory allocation, or even writing to the screen, it must use the API interface to request the kernel or subsystems to honor its request. The request may be handled by the subsystem, or it may require action by the kernel. If so, the API passes a request to the kernel and control is transferred into the kernel at ring 0. When the request has been satisfied, control returns up the chain, finishing where it began, at the application.

Unlike DOS, OS/2 applications are restricted from direct hardware manipulation. OS/2 acts as the "traffic cop" to coordinate the use of system resources.

Under DOS, a program has direct access to memory and devices using the Basic Input/Output System (BIOS) services. This is perfectly acceptable in a single-tasking system, because a program knows that it is the only one running and the system is under its command. In a multitasking system such as OS/2, this can be deadly. Many programs may be running in the system at one time, so synchronization of the system resources (memory, disk files, and display screen) must be managed.

Under the OS/2 system, applications *must* ask the system to operate on their behalf. Applications run at ring 3 and are not permitted to access the computer's hardware directly. They must use IOPL segments, subsystems, or device drivers for such operations. These levels of control are built directly into the 80286 and 80386 processors. Another type of control built into these processors controls data stored in memory, using address indirection.

Indirection (when referring to memory) means that applications do not know the exact address of the memory that belongs to them. What they see as an address in memory is a selector and an offset into a segment. OS/2 memory management still uses the segmented memory model as in DOS, but with a twist.

In DOS applications, a memory location is referenced by a program with an absolute address, which consists of a segment and an offset into the segment where the desired data resides. This cannot be done reliably in a multitasking environment, especially one with a memory overcommitment feature.

Memory overcommitment means that the operating system allows programs to use more memory than the system physically has. Data segments that are not currently in use are moved out to a disk file and then brought back in when needed. The technique is known as *swapping*. Code segments, however, are not swapped; they are simply discarded and reloaded from disk when needed. OS/2's memory management may also move segments around in memory to free up larger chunks of memory, eliminating small "holes" that were left behind by smaller than full-segment allocations. This feature is known as *memory compaction*.

In an operating environment that runs many programs at once, each using large amounts of memory, a program cannot be guaranteed an absolute memory location. Data may have been moved somewhere else or swapped out to disk. This is where selectors and indirection come in.

Indirection is accomplished by selectors, descriptors, and descriptor tables. The actual function of descriptor tables is very complex. Only the basics are covered in this discussion.

OS/2 uses two types of descriptor tables—the Global Descriptor Table (GDT) and the Local Descriptor Tables (LDTs). The difference between them is obvious from their names. The GDT contains pointers (or descriptors) to data and code segments accessible to all processes. Each process has its own LDT, however. This is what keeps data and code between processes separate and inaccessible. Any thread running may access a segment in the GDT, but it may access only its own LDT. Any attempted

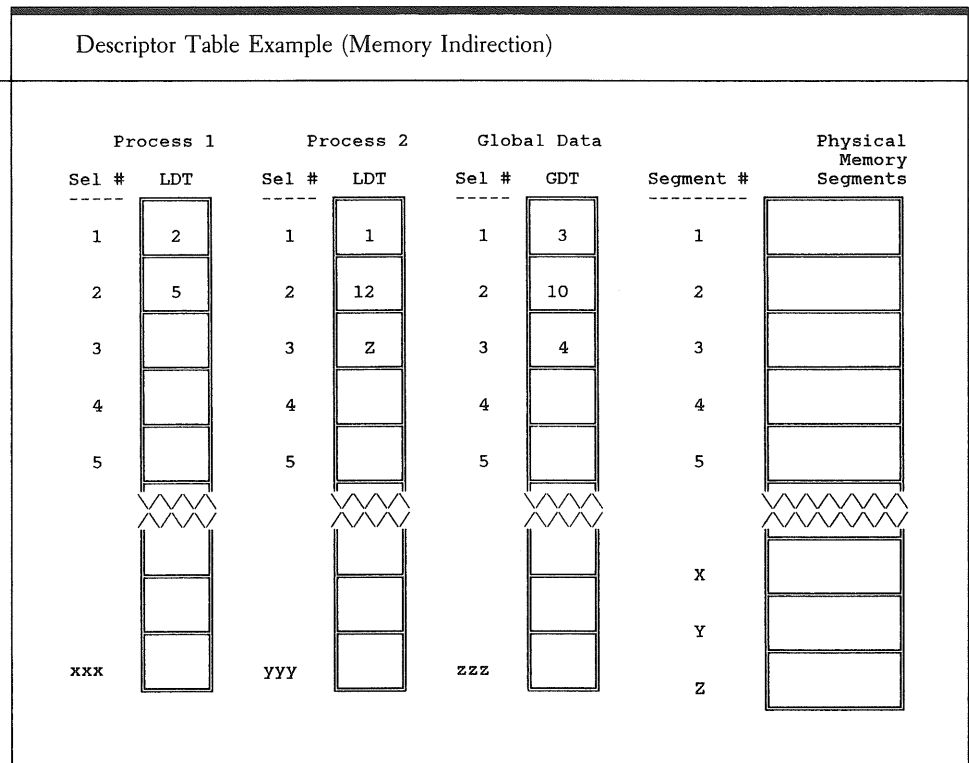
access to memory not in a program's own LDT or the GDT generates a protection violation and terminates that program.

The descriptor tables associate a selector with an actual memory address. The selector is an index into the descriptor tables. Each entry in the table contains a segment descriptor that contains all of the information about a particular segment. Figure 5.2 shows how this indirection works.

When an application needs to access memory, it uses a process similar to that used by PC DOS but with one key difference. In PC DOS a memory reference requires a segment and an offset. OS/2 memory references use a selector and an offset within the segment referred to by the selector. When an application makes a memory reference, the selector indexes into the application's LDT (or the GDT) for that particular segment descriptor. The descriptor notes whether the segment is currently in memory or swapped out to disk, identifies its real memory address when it is in memory, and determines what process it belongs to (among other things). Once the system knows about the segment, it tries to give the program what it needs.

If the segment is in memory, the data is retrieved for the application based on the segment's real address and the offset part of the memory reference provided by the

Figure 5.2



application. If it has been swapped out to disk, a “segment not present fault” will be generated, and the system will load the segment back into memory and provide the application what it wants.

This is an example of why applications may not use the absolute address of memory locations. Segments that are swapped out to disk are never assured of returning to the same spot in memory. They may also be moved during memory compaction. By using a selector rather than an absolute segment, an application may make a memory reference at any time. It is the operating system’s responsibility to make sure that the descriptor tables accurately show the locations of all segments.

OS/2 descriptor tables also monitor what sessions or processes own the various pieces of memory. Part of a descriptor is the session ID and process ID of the owner. If a program tries to use a selector that is not in its LDT or tries to access an area of memory that it does not own, a protection fault occurs. This is what keeps everyone honest. If an application attempts to access data of another program in an unauthorized way, the OS/2 “traffic cop” terminates the program.

OS/2 programs are protected from each other and, in some cases, from themselves. Such a protection feature does restrict the direct hardware access that many DOS programs enjoy, but that is the price to be paid in a multitasking environment. The functionality is not reduced, but applications must use a standard set of subsystems to accomplish their objectives. The benefit of such structuring is added standardization and cooperation between applications.

Device Drivers and Device Independence

OS/2 gives the programmer the power and flexibility to do many tasks that were either very cumbersome or impossible in earlier PC operating systems. One problem that application developers have always faced is deciding which devices to support in their application. Even if a software company wanted to support a specific device, such as an enhanced graphics display, there was still the problem of which vendors’ devices to display. Knowing that not all devices that perform the same function are compatible, many vendors decided to support only a certain list of devices. Others provided complex installation procedures to install the correct device drivers. Users also experienced problems when upgrading their hardware—new device drivers needed to be installed.

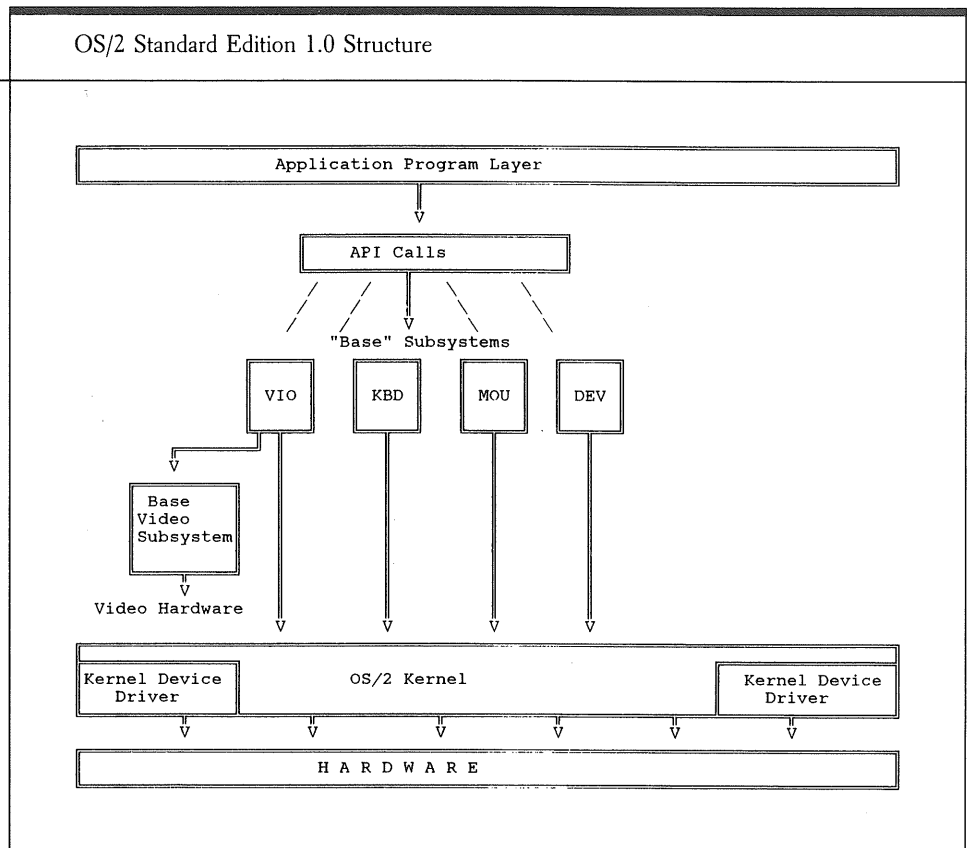
Often users have had to reinstall a software package to get it to recognize the new device drivers. Device drivers are special low-level programs that tie in very closely with the operating system core to allow programs to talk to the devices. They were written primarily by the software vendors to work with an application because there were no standard interfaces to the various devices. DOS provides only limited support for the devices that are available to the personal computer user. Either way, there are

always some devices that are not supported, and application developers must write drivers for all the devices that are supported.

Figure 5.3 illustrates the structure of subsystem and device driver interaction within the OS/2 1.0 system.

OS/2 applications are allowed access to any of the devices that the system supports by using the system APIs. In DOS, an application that wishes to use the IBM Color Graphics Adapter and also wishes to take advantage of the advanced features of the IBM Enhanced Graphics Adapter, if it is present, must supply device drivers for both. The software must provide interfaces to all of these drivers to take advantage of the hardware's features. The device driver is of no use if the application does not know how to talk to it. OS/2 Version 1.2 provides drivers for each device it supports, and others may be provided by hardware vendors. If these device drivers perform according to the rules of adding system extensions, they may be accessed just as any other part of the system, through the OS/2 APIs. Using the API calls, applications can operate

Figure 5.3



on the device without regard for its individual characteristics. This feature is known as *device independence*.

Device independence gives programmers the freedom to disregard the specific functions and capabilities of each device in the system and to code to a common set of routines. The translation to each device is done internally within the system, so the applications need not know what hardware is present. No longer does the software need to know the resolution of the display or what printer or disks are present.

Each OS/2 device driver must have a strategy routine and an interrupt routine. The device drivers are loaded at system initialization time, becoming extensions of the kernel. Device drivers run at ring 0, along with the kernel. A device driver has an initialization routine that runs when the driver is loaded. The code usually sets up static data areas for the driver and communicates with its device to tell it to initialize and perform any action necessary to prepare it for operation.

Once initialization is complete, the driver returns control to the system, other drivers are loaded, and the system finishes its initialization process. The device driver sits in a dormant state until it receives a request packet from the kernel. Now the strategy routine is invoked. A strategy routine is nothing more than a routing routine that receives the information from the kernel and transfers control to the proper part of the interrupt routine. The interrupt routine performs the actual work. Once complete, control, along with the data, is passed back to the kernel.

Applications may use the OS/2 device drivers through the APIs, which start a chain reaction of calls down through the kernel to the hardware. Isolation of the application from the hardware relieves applications from concern about the hardware that is present. The APIs map to a set of kernel calls that are translated into the hardware. As long as programs use the APIs, they are assured of portability to other OS/2 systems or versions, regardless of the hardware.

Figure 5.3 shows how applications communicate directly with the OS/2 subsystems. They are implemented as a set of dynamic link libraries, which include the Basic Video Subsystem, Keyboard Subsystem, and Mouse Subsystem. The subsystems run at privilege level 2 or 3. These subsystems are called when a program uses an API that requires services from the device for which the subsystem is responsible.

Once a subsystem is given control, it makes a call into the kernel for the services it needs. The kernel may then make a call into a device driver or satisfy the request itself. Memory allocation is a good example of this. There is no special device driver for memory; memory management routines are built into the kernel. The subsystem may also make a call directly to the device driver through the kernel. In reality, all device driver request packets get to the driver from the kernel, so a "direct" device driver call really just sends a request to the kernel to pass the request packet to a specific device driver. The device driver then responds to the request made of it and control returns up the chain, ending where the call originated, the application program.

To sum up, the application program uses the APIs to make requests of the system. The subsystems satisfy the request by making a call into the kernel. The kernel satisfies the request using one of the device drivers installed into it or using its native services.

Handles

Indirection is a focal point in this multitasking system. Handles must be used rather than specific addresses for elements such as memory and devices. This allows the system to manipulate the hardware and coordinate access between processes.

Although OS/2 is not the originator of this concept, it makes extensive use of handles. A *handle* is a tool for indirection that applications use to manipulate the computer system. The OS/2 system has handles for nearly everything: devices, files, memory, and windows are examples. Handles are used just like actual addresses, but because applications must request system services through APIs, handles get resolved by the APIs to the actual addresses to serve the application.

All handles do not contain the same information. A handle contains the information that the system needs to access a piece of hardware with respect to the requesting process. For example, a handle to memory is the selector we have been discussing. A memory address, as a program sees it, is this handle, which is the selector, and an offset into the segment the selector represents. A file handle contains information specific to that process and how it accesses the file.

Because access to the OS/2 system resources is managed with handles, each program may act as though resources belong only to it. It is up to the operating system to coordinate the access to its resources. Programs need not concern themselves with whether the segment needed has been swapped out or moved or what the addresses of different devices are. Once the program obtains a handle for that resource via an API call, the OS/2 subsystems take care of all of these details.

A well-behaved application must not attempt to circumvent this mechanism by trying to use absolute addresses; otherwise, it ends up with unpredictable results or a protection violation. By letting OS/2 handle this detail, applications are free to concentrate on their special functions. They do not have to worry about the status of the resources they need; OS/2 is the application's servant.

This approach has many advantages. The most important is that the use of handles gives applications the freedom to receive input from a variety of devices and send output to a variety of devices with a minimum of code. Because the handle is a level of mapping from the application level to the system level, the application is further isolated from the hardware. Common routines may be written to input or output from or to devices without knowing anything about the device. Only a handle must be passed to a common output routine. The common output routine does not need to know whether it is sending output to a printer, screen, or file.

APIs

The Application Programming Interface (API) routines are the building blocks of all OS/2 applications. APIs enable applications to communicate with the operating system surrounding them. Each API is a far call into a set of dynamic link libraries that collectively make up the OS/2 subsystems. These subsystems allow the applications to control their environment and use OS/2 services.

The names of most APIs that deal with the internal systems of the base begin with the prefix *Dos*. For convenience, this book refers to them as the *Dos calls*. The name of each *Dos* call is indicative of the function it performs, a means of adding consistency and readability to source programs. It is easy to understand a call labeled “*Dos-CreateThread*.” All APIs conform to a set of standards in the type of data they pass and the names they use.

There are *Dos* calls for memory management, thread control, interprocess communications, and all the OS/2 system services. All system services must be requested via APIs. The OS/2 dynamic link feature allows programs to call and share routines in the system. All OS/2 APIs are implemented with this feature. The system is easily extendable by programmers; new APIs may be added and placed in “*dynalink*” libraries for all programs to use.

OS/2 is a powerful system for application developers. It provides a great deal of flexibility and a consistent, easy-to-use interface to system services through APIs.

Dynamic Linking

As time goes on, many useful routines are programmed and may be applicable to other programs. Many of these functions have been grouped together into sets of libraries. They are stored in a *.LIB* file and are callable by programs set up to use them. When a program requires the use of one of these functions that is not directly in its own code, it makes an external reference to one of these library functions. This is usually accomplished with an *EXTERN* declaration, along with the function’s prototype (what type of data the function expects).

Many compilers (and the OS/2 Programmer’s Toolkit) come with sets of header files that supply function prototypes called *headers*. All you need to do is include the right headers, and the functions are there for you to use. You are saying to the compiler, “This is a reference to something external to this module. I’ll tell you about it later when I go through *LINK*.”

When a program making these external references is compiled, these function calls are not fully resolved. That is, the code to execute that particular function is not inside the module. Only a reference to it is. The keyword *EXTERN* simply tells the compiler

that you know the function is not there, and you know where it is. Otherwise, the compiler flags it as an undefined symbol. When the compiler encounters these EXTERNALs, they are marked in the object (.OBJ) file as an external reference. When the program is linked with the .LIB file, these references are resolved with the code in the library. Now the functions used from the library are incorporated into the application's executable (.EXE) file. This is called *static linking*. The external routines are statically linked into the executable program and become part of that program.

The OS/2 dynamic link feature allows applications to make external references to functions contained in a special type of library, the Dynamic Link Library (DLL). As the name implies, the DLL operates differently from a static library. In using the dynamic link library, an application need not bind the function to its executable file. The function call is left as a special type of external reference in the executable file.

When an .EXE file containing these special references is loaded, the OS/2 loader recognizes where the function resides (which DLL module contains the function) and loads that module. If the DLL already resides in memory, a reference is generated so that the .EXE can use the copy already in memory. The result is smaller .EXE files that load faster and take up less memory.

DLL libraries are fundamentally the same as .LIBs, but they feature several important differences. A DLL file is structurally a combination of a .LIB file and an .EXE file. It is simply a collection of functions (like a .LIB), but it has an .EXE-type header, so it may be loaded by the OS/2 loader. DLLs are special libraries that are not incorporated into the executable file. They are separate files that are loaded when called for by the application programs. When a DLL is loaded, an LDT entry is created for the process requesting the DLL's functions. This way the process may make a far call to a routine residing in a segment not belonging to it without generating a General Protection fault.

Once a DLL has been loaded into memory, the functions inside it are accessible to any program running in the system without requiring another copy of the DLL to be loaded. This saves memory when many applications need the services of common functions. This feature is the basis of the OS/2 subsystems. Only one copy of the DLL needs to be loaded for all programs to have the use of its services. With static linking, each application has the library routines inside its .EXE file. Applications using dynamic linking only need references to the function inside their .EXE file.

The reference to the entry point within the DLL is accomplished with a special .LIB file containing records to resolve the external references in the program at link time. These records contain the information for the program to access the function in the DLL. When a program is linked with this .LIB file, reference records are generated and placed in the .EXE. This satisfies the external reference for the linker.

When a program needs the function contained in the DLL, the reference record is interpreted, and OS/2 loads the DLL according to the information in the reference record. The address of the function is resolved, allowing the program to make the far call into that routine.

If the DLL is currently in memory (after it's loaded by another program), the system does not load another copy. The system simply resolves the call for the application to allow it to use the DLL's functions. When the last program using the DLL is finished, either by termination or by explicitly freeing the module, the DLL is removed from memory until the next time it is needed.

Subsystems

The subject of OS/2 subsystems is very complex. The OS/2 subsystems may be thought of as the routines that separate the API call, the kernel, and device drivers in the system hierarchy. They perform most of the translation of information from an application program to the low level required by the hardware. They take the place of the BIOS services that performed these services under DOS. The subsystems handle the coordination between processes with respect to the I/O hardware. The OS/2 base system has subsystems for the keyboard, video hardware, and mouse. They are implemented primarily as a set of dynamic link libraries.

When an application makes an API call to access the video input/output (VIO) subsystem, for example, the API is a routine inside the VIO subsystem DLL. This routine translates the application's request into request packets to the kernel and its attached device drivers to perform the requested function. The subsystem takes the information from the application and uses status information about the current state of the system (such as which session is in the foreground) to pass request(s) into the kernel to perform the function.

This level of isolation is necessary because an application may have the state of its resources changed at any time. Applications may be switched from foreground to background or vice versa. Segments may be moved or swapped in or out of memory. Each session in the OS/2 system has its own logical set of devices that is managed by the subsystems. These devices may be treated by a program running in that session as its own. The subsystems manage the devices in such a way that they become "virtual" devices. That is, the application may request that operations be done with the devices, but the device may not respond at that instant because some other session may be in the foreground.

When the session comes to the foreground, the action has been taken. The action was performed in that session's virtual device while it was in the background. It becomes physical reality when that session comes to the foreground. If screen I/O is being performed to a window that does not have the focus in the Presentation Manager session, the display changes as the I/O is performed for those areas still visible.

Each of the OS/2 base subsystems is a DLL that works with the individual devices present in the system. With DLLs, it is easy to add extensions to OS/2 by adding new DLLs, and the Presentation Manager shell does just that. It adds more subsystems

(using DLLs) to OS/2 to allow more advanced functions with the same set of devices. Extensions may be added to OS/2 by adding DLLs to perform new functions. Anyone may add system extensions with new DLLs.

HPFS and IFS

OS/2 Version 1.2 introduces the installable file system. The installable file system (IFS) mechanism allows multiple file systems to coexist in the same system.

One file system in addition to the standard FAT file system is included with OS/2 Version 1.2. This system is called High-Performance File System (HPFS). The user is asked at installation time whether to install HPFS.

Some of the functions that HPFS affords the system are extended attributes for files, a large file cache, long file name support, and better overall performance.

HPFS can be used for the boot partition or any extended partition in the OS/2 system. This feature enables users to migrate from the FAT-based file system to any number of different file systems.

HPFS is only one implementation of the IFS mechanism. As time goes on you will likely see other file systems for OS/2 appear (or you may even write one yourself).

Advanced Utilities

As each new release of OS/2 appears on the scene, new and advanced utilities and commands come with it.

In OS/2 Version 1.2, the most prominent new command is PSTAT. PSTAT is very similar in function to the UNIX® command, **ps**.

PSTAT is a utility that displays the status of all the processes in the system. Using this command, you can view the state of any or all processes; see whether they are running, blocked, or frozen; and see the different resources (like semaphores) each is using.

In addition to PSTAT, Version 1.2 includes many extensions to the commands you have come to know such as **FORMAT**.

With HPFS on the scene, in Version 1.2 you need to specify what type of file system you wish to format a partition with.

As you have already seen, the online command reference is at your fingertips to explain all these new commands.

Summary

The base portion of OS/2 provides the system services needed to run applications. These services include the I/O services, memory management, and program isolation.

OS/2 BASE SYSTEM FEATURES

Using these facilities creates new challenges but affords you new functions to accomplish your tasks.

Dynamic linking, segment selectors, and the protection mechanism afford OS/2 applications more freedom to run as if they were the only application running in the system. These mechanisms enable this freedom by preventing programs from bumping into one another. This protection helps applications to exploit the full function of their environment.

6

OS/2 Presentation Manager Functions

The Presentation Manager session is the most visible difference between the first two OS/2 releases. The Presentation Manager interface offers many advantages over a full-screen text interface. The most important of these are the graphical user interface and the use of multiple windows that can be displayed simultaneously.

The Presentation Manager interface is more than just a graphical user interface. To the user, it provides impressive graphics and menus that reduce keystrokes because many things can be done with the mouse. It provides a much easier and more pleasant way to interact with the computer.

To the programmer, it provides a new way of writing programs and a new way for programs to communicate with the hardware, with users, and with each other. Along with the function of the base system, Presentation Manager programs use windows, messages, hooks, and a myriad of objects.

Types of Applications

Four types of applications are available in OS/2 Standard Edition. Because the features and functions available to each type differ, the objectives of each program must be analyzed to ensure that its objectives match the environment. The four types of programs are

- DOS compatibility mode

OS/2 PRESENTATION MANAGER FUNCTIONS

- OS/2 protect mode full-screen text
- VIO-windowable text
- Presentation Manager
 - a. AVIO
 - b. Graphics

The DOS compatibility mode is usually used for programs written for DOS. The DOS compatibility mode in OS/2 Versions 1.0 and 1.1 has a compatibility level equivalent to DOS 3.3, whereas OS/2 Version 1.2's compatibility mode is at the DOS 4 level.

The "DOS box," as it is sometimes called, is exactly that. It is a box within the OS/2 system reserved for DOS programs. It is not required for operation; users may set up the configuration to exclude this box.

Applications running in the DOS box are written either to the INT architecture of DOS or to a certain subset of the APIs known as the Family APIs (FAPI). Applications conforming to these guidelines are assured of execution in the DOS box.

An OS/2 full-screen application is equivalent to an application that has been written to run under OS/2 Version 1.0. Applications running in a full-screen session do not have access to any of the Presentation Manager services. The windowing and graphics features are not available, nor may full-screen programs communicate with Presentation Manager programs using the Presentation Manager's messages. They may communicate only using the OS/2 interprocess communications facilities, such as pipes and queues.

Most of the OS/2 APIs are compatible with and may be used in the Presentation Manager session. A few, however, may not be used in Presentation Manager programs, and those programs that do use these restricted APIs must be run in a full-screen session. Applications that do not use these restricted APIs may be VIO windowable. These are programs that are not specifically programmed to run under the Presentation Manager shell but may be run in a window. The list of restricted calls may be found in Appendix D.

VIO-windowable applications are not written to take advantage of Presentation Manager features; as such, they may not use services that include custom dialogs and menus. The system understands when an application is VIO windowable, and the Presentation Manager shell does its best to do the window housekeeping for that program. The shell runs the program in a plain window with limited functions, such as scrolling and window sizing. This type of program may not use the Presentation Manager message-passing mechanism or any of the advanced graphics functions. It is only *windowable*.

There are two types of Presentation Manager programs: AVIO and graphics. AVIO programs are text-based programs using Presentation Manager APIs and base APIs. Graphics programs may use all the Presentation Manager facilities. Programs written to take advantage of this are given these advanced functions simply by adding some function calls. No knowledge of the hardware specifics is necessary to achieve impressive-looking applications.

What Is SAA, Anyway?

The Presentation Manager interface is the OS/2 implementation of Systems Application Architecture (SAA). SAA is a set of guidelines for application presentation and use on IBM computer systems. It outlines how applications should look and behave. SAA states that applications conforming to these standards are ensured portability to other IBM SAA systems.

In many instances, compliance with the SAA guidelines is built in to Presentation Manager programs. Many of the APIs adhere to the SAA guidelines, and because Presentation Manager programs must use the APIs to function, compliance with SAA in these respects is automatic. Many other aspects of SAA are completely up to the application designer. Examples include use of colors within programs, functions the PF keys perform, and ways applications position information on the screen.

A few final words on SAA: The Presentation Manager interface is the platform for the SAA Common Programming Interface and Common User Access, and it is the base on which all SAA applications grow. In the past, many applications had to be rewritten for different operating environments. SAA conformance is a sure way to minimize the amount of work necessary for porting an application to another SAA system. However, SAA is a set of guidelines—not rules—and programmers are still free to implement functions as they wish.

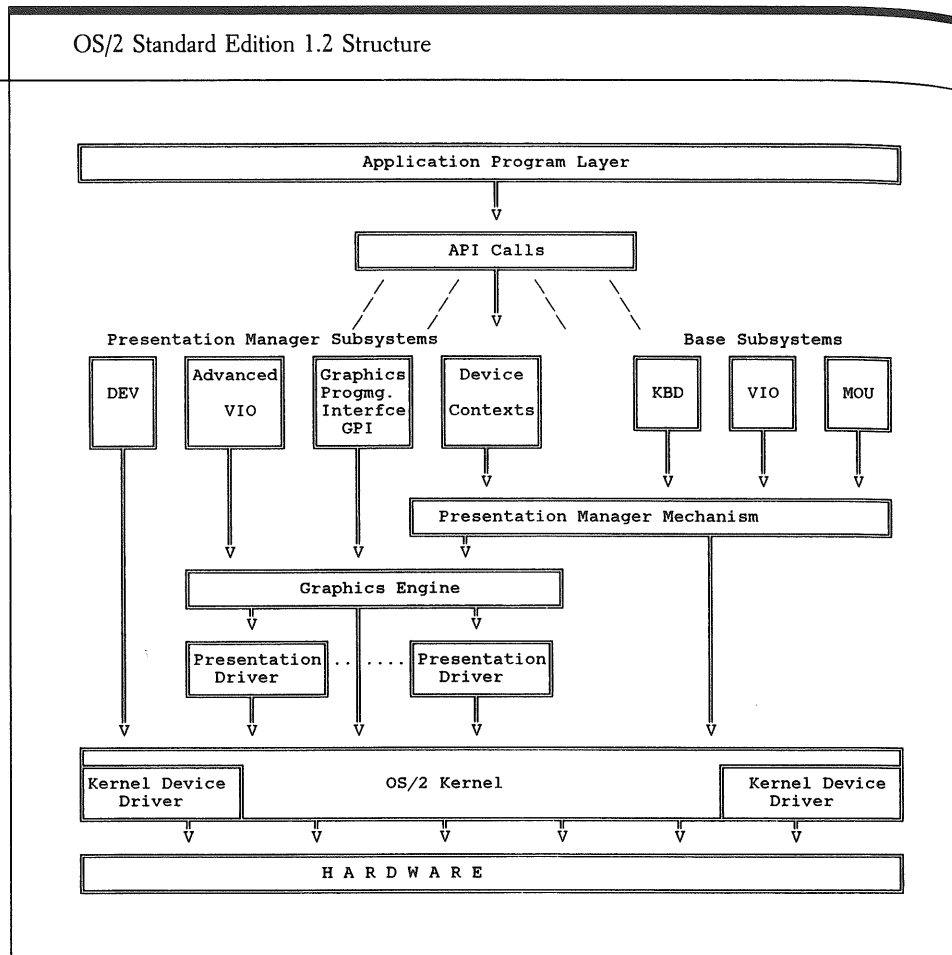
Presentation Manager Device Drivers

The way device drivers are implemented in OS/2 Standard Edition Version 1.2 is not much different from other operating systems, but there are a few more layers of abstraction in this system. These extra layers provide further independence from the hardware.

The Presentation Manager facilities are much more powerful than those of its predecessors. To provide device independence with advanced capabilities, the operating system must do much more for the application program. New components that provide this advanced function are referred to as the *presentation drivers* and the *graphics engine*. These layers are shown in Figure 6.1.

Figure 6.1

OS/2 Standard Edition 1.2 Structure



When an application needs to use system services, it simply makes an API call into the subsystem layer. In addition to the base subsystems, the Presentation Manager subsystems include graphics (the GPI package), advanced video input/output (AVIO), and device context management.

The device context area handles the translation and coordination of access to the physical devices. Every device that is to receive output from the Presentation Manager needs a device context to perform the actual translation of device-independent commands into packets that can be understood by the drivers controlling that device. This area of the system handles the logical coordination of the screen, for example, so that all screen I/O goes through it. When a program requests that a window be displayed on its behalf, or a program requests that a graphic picture be drawn, this subsystem controls the logical end of it and tells the physical device (or driver) what to do.

The AVIO component supports rectangular character presentation spaces, in contrast to the graphical presentation spaces of the GPI. Functions are provided to construct the presentation space, as well as to display the presentation space in a window or windows.

VIO-windowable programs are those that use the AVIO component but are not full-fledged windowed applications. They can be run in a basic window and have some limited window functions, such as sizing and scrolling, but they cannot take advantage of the more powerful window features, such as menus and advanced graphics. Simply put, a window-compatible program is a hybrid between a windowed and a full-screen program.

The GPI interface is where the graphic object creation and manipulation are done. Each API for these functions begins with GPI and deals with all of the graphics functions. When the application wants to generate a line, a call is made to one of the GPI routines, which calls this GPI subsystem.

The GPI subsystem talks directly to the graphics engine to manage graphics output. The graphics engine is the next level down in the request for data display for Presentation Manager programs. It manages the output in the Presentation Manager screen group. It is basically another subsystem at a lower level than the API subsystems that talks to the kernel and a special set of drivers, called *Presentation Drivers*.

The Presentation Manager shell adds some new subsystems with new DLLs. In addition, there is a set of low-level drivers that interact with these DLLs. Depending on which services are needed, the request can take different paths. The simplest case is the *Dos request*.

The Dos type API requests services that are, for the most part, built into the kernel. They are either native to the kernel or are in the kernel device drivers that become extensions to the kernel when it is loaded. Examples of these are memory management calls (DosAllocSeg, DosFreeSeg), thread and process management (DosCreateThread, DosKillProcess), and input/output calls to devices via the DosDevIOCtl call. Because the routines that are needed to process these requests are either native to the kernel or contained in kernel device drivers, the Dos portion of the API subsystems makes requests directly to the kernel. The group of APIs that make requests into the kernel also includes the Dev calls, a special set of APIs that request the services of device drivers directly through the use of I/O control (IOCTL) packets.

The other base subsystems—the keyboard (KBD), mouse (MOU), and video input/output (VIO)—must pass requests through the Presentation Manager mechanism for the services they require. One of the Presentation Manager subsystems, the device context part, uses the same path as well. The Presentation Manager mechanism manages all resources at the application level. It makes sure that windows do not destroy others, that keyboard input goes to the focus window, and that mouse input goes to the window for which it is intended. The Presentation Manager mechanism manages the input and output resources and coordinates all applications. It uses a structure

known as the *message architecture*, which is mentioned briefly here and discussed in depth in Chapter 8. For now, it is important to know that the Presentation Manager mechanism is responsible for a level of translation and coordination between the application program's API calls and the lower levels of the system.

Once the request for services is processed in the Presentation Manager mechanism, the translated requests go either to the kernel directly or to the graphics engine, depending on the nature of the request. These levels of translation are necessary to provide both the hardware independence and resource management for the operating system. Many calls, such as certain memory requests, thread and semaphore control, and other such kernel services, do not need to pass through the Presentation Manager shell. But others, such as ones that use any type of I/O, need to be managed by the Presentation Manager mechanism. This uses the graphics engine for its presentation services.

The graphics engine is a special device driver. Unlike regular device drivers, it is not part of the kernel. This is where the Presentation Manager makes the biggest departure from the first version of OS/2. The graphics engine contains the routines to turn all the graphics calls into reality. It accomplishes this in one of several ways:

- First it tries to satisfy the request by simulation. If the graphics engine is able, it responds to the requests made of it using simulation routines it contains. Many graphics requests may be simple enough that the graphics engine can perform the operation without any further translation or kernel interaction.
- If the graphics engine cannot satisfy the request via its simulation routines, it may pass the request to the presentation drivers. Special routines that are not part of the kernel take graphics engine requests such as arc, line, bitmap, and string manipulations and convert them into series of requests the kernel can satisfy. These requests cannot be satisfied by the graphics engine alone. The presentation drivers may be able to satisfy the requests from the graphics engine by using routines they contain. In this case, the requests need not go to the kernel level.
- The graphics engine also has the power to pass requests directly to the kernel, bypassing the presentation drivers. In some instances, the requests are made directly to the devices. There are many different combinations of graphics functions that may be used and their requests for work to be done by the lower levels of the system on their behalf may be satisfied in a variety of ways. The method chosen by the system for each request is based primarily on performance reasons.

Once at the kernel, all requests have a common format. If the request came from the graphics engine or presentation drivers, it becomes part of a series of requests that together compose the desired graphics function. The calls that come from either the Presentation Manager mechanism or the DEV subsystem are usually part of a smaller series of requests that performs a less complex function but an important one, never-

theless. The kernel satisfies the requests made of it and returns control back up the hierarchy, returning to the application level where the high-level API call originated.

The internal model of OS/2 with the Presentation Manager differs in complexity from Version 1.0, but not in basic structure. At the application level, there is little difference. There are more APIs, but that is all applications must contend with. They need not be concerned with the underlying structure just outlined. This is true device independence. All the translation to the hardware features, capabilities, and limitations is done within the operating system. As long as the system supports the attached hardware, applications may take full advantage of all features transparently.

The Presentation Manager Session

Running OS/2 Version 1.2, there is only one Presentation Manager screen group. Version 1.0 used the Program Selector screen group and 12 user screen groups. In OS/2 1.2, the screen group that was used for the Program Selector is now the Presentation Manager screen group, and the other 12 remain text mode, full-screen sessions.

When developing an application for the Presentation Manager, programmers must understand some characteristics of the session in which it runs. The main point is that all physical I/O should be done through the Presentation Manager. There is a set of base calls, such as certain Vio calls and Dos calls, that are not supported in the Presentation Manager screen group. These restricted calls are listed in Appendix D. For the most part, they deal with direct hardware I/O requests and registration of new subsystems and monitors.

Because of the complex management required of the Presentation Manager shell, direct I/O functions, such as writing to the physical screen buffer or installing keyboard monitors, may not be done in that shell. In the other full-screen sessions, such tasks may be done. In most cases, the Presentation Manager mechanism provides an alternate method through one or more of the Presentation Manager APIs.

APIs

There are several major categories of APIs in OS/2:

Dev calls	Device control functions
Dos calls	Kernel (memory, threads, signals, and so on)
Gpi calls	Graphics programming interface
Kbd calls	Keyboard subsystem
Mou calls	Mouse subsystem

OS/2 PRESENTATION MANAGER FUNCTIONS

Prf calls	Desktop Manager profiles
Spl calls	Spooler functions
Vio calls	Video input/output subsystem
Win calls	Window management and user shell

Each set of calls can generally map to one of the OS/2 subsystems. All applications in the operating system use APIs to talk to the subsystems, which in turn make requests of lower-level system services on behalf of the application. The major additions to the API list for Version 1.1 and 1.2 are the Win calls and Gpi calls.

Win Calls

The Win calls are all of the APIs that begin with the *Win* prefix, such as WinCreateWindow or WinInitialize. These calls provide the functions necessary for applications to interact with the Presentation Manager. There are calls for window creation, manipulation, error handling, status checking, and destruction. The Win calls set includes functions providing all of the housekeeping necessary for applications to use Presentation Manager services.

The Win calls give programs access to the powerful Presentation Manager windowing features. They provide applications with the facilities of messaging, resources such as dialogs and menus, complex window management tools, and advanced methods of user input. In past operating systems, programs had to have access to input functions built into them or linked in from external libraries. The Presentation Manager structure of the Win calls provides a powerful set of I/O functions and the standardization necessary to ensure portability and freedom from device restrictions.

Many of the Win calls emulate functions of their Dos call counterparts to allow the Presentation Manager functions to perform some of the functions handled by the base subsystems. The reason for this "duplication" is that direct I/O is often not advisable or possible in the Presentation Manager session. Using the Win APIs for tasks such as obtaining keyboard key states (WinGetPhysKeyState) or processing mouse input in an application (using the WM_MOUSEMOVE messages rather than Mou calls) the Presentation Manager shell can manage resources more safely while not reducing access to this functionality. Many of the base calls for these functions are supported in the Presentation Manager session, but for consistency, Win calls should be used as much as possible for I/O functions.

Gpi Calls

Gpi calls are used for all graphics programming. There are APIs for graphics primitives, metafile and bitmap storage and manipulation, and graphics output. Some Gpi calls associate program objects with areas on the screen and other output devices, such as

files or printers. These calls provide applications with the facilities to generate graphics objects from the simplest lines to the most complex figures. Other calls manipulate complex pictures and facilities for storage of any type of picture. The GPI routines contain functions for creating areas to draw on, primitives to draw with, and fonts to write with.

GPI functions operate on presentation spaces, each of which is associated with a device context, allowing the GPI to function without knowing the specifics of the final device. Various modes of presentation spaces and graphics manipulation are available to provide a wide range of graphics output, from simple to complex.

In addition to the windowing and graphics calls, other APIs give programs access to additional functions within the Presentation Manager. Dynamic Data Exchange (DDE), message hooks, fonts, and accelerators are examples.

Windowing

In the Presentation Manager session, the window is the most fundamental object; it must be understood before any of the advanced features can become useful. If you are unclear about what windows are and how they tie in with this system, please review the windowing descriptions in Chapter 4 before continuing with this section.

Most applications running in the OS/2 Version 1.2 operating environment use windows as their primary medium of input and output. An application usually has at least one window that serves as the application's interface to the user. Applications have the power to control everything that happens to their windows. The user employs the system's tools, such as the mouse pointer and keyboard, to manipulate and communicate with the window. The way the window responds is completely up to the program controlling it.

The simplest way to describe a Presentation Manager program is to say that it looks for everything it is interested in and acts on it. Anything else is passed on to the system. This is a departure from what many programs available today are designed to do. Programs written for earlier systems need to be aware of everything the user is doing. These programs must understand all the actions a user may take. All key combinations, mouse operations, and so on, must be taken into account. Output must be handled in a very detailed fashion. The Presentation Manager changes the way applications handle input and output.

The Presentation Manager windowing facilities are complex and extremely powerful. Applications do not need to be concerned with the manipulation of their windows or whether the windows are visible, hidden, large, or small. Programs are concerned only with their own processing and the input and output methods they choose to use.

Programming for the Presentation Manager system is akin to object-oriented programming. The object here is a window and the procedure that controls its behavior.

As programs run, their windows receive messages from different parts of the system and they must respond to them. The message does not go to the window per se, but to the window's procedure. The window is identified by its handle, and when a message is addressed to that handle, it really goes to the procedure for that window. The object is manipulated by the data contained in the messages sent to it. Any messages that the object is not concerned with may be passed on to the system.

The default processing provided by the OS/2 Presentation Manager takes care of all the tasks programs do not wish to handle. Several special window procedures are built into the Presentation Manager subsystems to handle the default processing of all messages. The main one is `WinDefWindowProc`, which takes care of the processing of any message a window does not wish to handle. Deeper in the Presentation Manager structure, there are many messages that a program needs to have handled by the default procedure, but with a little modification. This is also possible by executing some function when a particular message is received and then sending it to `WinDefWindowProc` when the program is finished with it.

Window programming is a very powerful, yet easy-to-understand, concept. Section Four contains programming examples and illustrations that make these functions easy to master.

Messages and Queues

Presentation Manager messages carry data and instructions. Each message contains an addressee, the message identifier, and the message contents. All the work done in the Presentation Manager session is accomplished with messages. For example, when it is determined that a window needs to be updated, it sends an update message, which may trigger messages to redraw certain parts of windows and inform others that all, or parts of them, have been covered, moved, or changed in some way.

Messages get to their destinations by way of the message queue. Each application has at least one message queue that receives the messages destined for the thread that owns it. When messages are sent through the Presentation Manager mechanism, they arrive in a queue. The thread owning the queue must remove the message and process it.

The Presentation Manager uses messages to prioritize work that needs to be done in its session and to coordinate access to system resources. Rather than waiting for user input and reacting to it, applications look for messages and act on them. No longer does a program need to redraw a whole screen when it thinks something has changed on it. The Presentation Manager tells it exactly what has been done via messages; the program only needs to act on those aspects that have been affected.

Even if a program or window is not the receiver of a particular message, it *must* be processed in some way, usually by the default procedure. When messages go un-

processed, unpredictable results occur. The programming examples included in this book include techniques to ensure that no messages are lost.

Resource Management

Objects besides windows are available to Presentation Manager programs. Icons, bitmaps, accelerator tables, and menus may all be manipulated by the Presentation Manager shell. Such objects are called *resources*. They may be created and maintained independently of programs. In addition, resources may be used by several applications. They are created and maintained separately, or they may be created dynamically during program execution.

Static resources are created using a resource definition (.RC) file. Such a file contains the names of resources and the attributes that control their behavior. The file consists of sets of resource statements that define the type of resource and how they are to appear and act. Each type of resource has its own definition syntax.

The types of resources available are

- Accelerator tables
- Dialogs
- Fonts
- Icons, pointers, and bitmaps
- Menus
- String tables

Each resource serves its own function in the Presentation Manager.

Several types of resources (ICON, DIALOG, and FONT) are created using the editors provided in the OS/2 Programmer's Toolkit. Once created, resources reside in their own special files that can be referenced in the .RC file and in the application program. Resources such as dialogs and menus may also be defined directly in the .RC file.

Each resource has its own definition syntax that defines it to the application program. Every instance of a resource must have its own definition statement.

Once the resources are created and defined in the .RC file, they must be bound to the application's executable file to build a complete Presentation Manager application. This task is performed by another toolkit program, the Resource Compiler (RC.EXE). The Resource Compiler is a special compiler that operates in two steps, similar to the operation of a high-level language compiler.

In the first stage of compilation, the resource compiler reads the .RC file, checks the syntax of the resource statements, and generates a file (.RES) that is similar to an .OBJ file that is the output of a language compiler.

The second stage of the resource compilation procedure is also accomplished with the resource compiler. In its second pass, it combines the application's .EXE file, which is the output from the OS/2 linker, with the .RES file that was generated in the previous step. This creates a final executable file for use with the Presentation Manager. The procedure is outlined in Figure 6.2.

An alternate procedure, which will be explained later, allows resources to be built into a dynamic link library.

Hooks

Hooks are another powerful tool in the Presentation Manager session. A *hook* is a mechanism by which Presentation Manager applications may detect an event and act on it before it gets to its destination for processing. Monitors are not supported in the Presentation Manager, but hooks serve the same purpose.

Every user input message in the Presentation Manager session must pass through at least one queue. Each queue in the Presentation Manager may be hooked. Each queue may have one or more hook procedures attached to the front of it so each message headed for that queue must pass through the hook before it gets into the queue. A hook is nothing more than a procedure attached to a message queue.

Hooks are installed using the API call `WinSetHook`. This call will take a procedure and a handle to a queue and set it up so that a message passes through the hook procedure before it gets into the queue. Multiple hooks may be installed on a single queue. They will be called in reverse of the order in which they were installed. When a hook is installed with `WinSetHook`, it is placed at the head of the hook chain for that queue. So, the last hook installed is the first to be called.

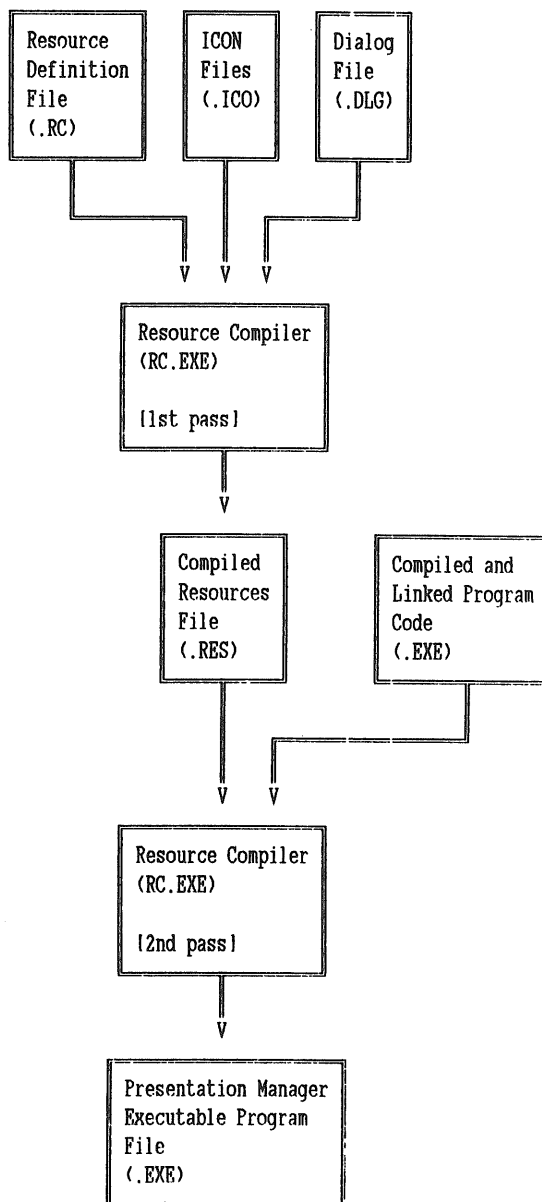
Two types of hooks may be installed, depending on which application's messages they process: an application hook and a system hook.

An application hook is hooked to an application's message queue. Using this type of queue, all messages going to an application may be monitored before getting to the application's queue. The hook procedure(s) applies only to the thread that owns the queue to which the hook is installed. This type of hook procedure is usually contained in the application's .EXE file but may reside in a DLL.

A system hook is one that attaches itself to the main Presentation Manager input chain. It is a public procedure that applies to any application running in the Presentation Manager session. As such, a system hook literally becomes part of the system. Because it is part of the system, the system hook must be accessible to all Presentation Manager programs and *must* reside in a DLL.

Figure 6.2

Compiling Resources with Executable Code



Hooks play a useful role in that they return function lost by the inability to register monitors in the Presentation Manager session. Using hooks allows applications to monitor Presentation Manager messages. Hooks may also trap messages destined for their message queue(s) and take some action on them before they reach their destination. Hooks are also useful in providing contextual help (HK_HELP hook).

Graphic Object Manipulation

Nowhere is the statement "A picture is worth a thousand words" more true than in the presentation of data on a computer screen. Quite often, data is made easier to read and understand with graphics. Pictures are also an international language. Graphics often convey ideas and information more effectively than words.

The Presentation Manager graphics functions provide a rich set of routines to enable applications to display information in a variety of ways. Graphics are displayed in Presentation Manager windows, just as pages in a book. Using the features of device independence, a standard set of graphic functions may map into and take full advantage of the available display hardware. In addition to handling the device mapping, the Presentation Manager mechanism can also clip the graphics to the edges of their presentation areas and manage similar aspects of the graphics presentation.

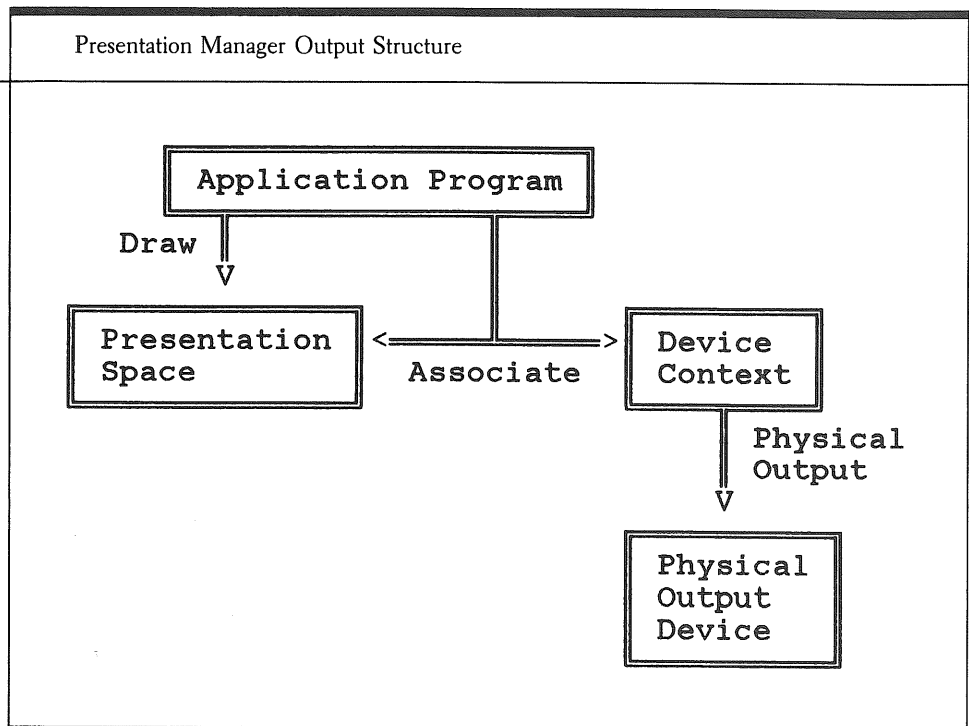
With these powerful features comes a bit of housekeeping. Programmers must be aware of several levels of translation when they work with graphics in the Presentation Manager session. The two main translators are the device context and the presentation space.

A *device context* maps the pictures in a *presentation space* to the output device. The output device may be a display screen, printer, plotter, or any other device the programmer uses to display graphics. The device context has the device-dependent information to turn the device-independent instructions contained in the presentation space into the actual display on the output device.

The device context gets its instructions from a presentation space. The data to be drawn in the presentation space is completely device independent. At the presentation space level, the application does not know (and does not need to know, at this point) what device will display the data. Once the presentation space is associated with a device context, the device mapping takes place. This is shown in Figure 6.3.

At any point in time, one presentation space may be associated with only one device context, and one device context with one presentation space. However, if an application has drawn a picture on one device and wishes to draw it on a different one, all it needs to do is to disassociate the presentation space from the first device's device context and associate it with the second device's device context and redraw the picture. Once this is done, the presentation space may be reassociated with the first device context.

Figure 6.3



The Presentation Manager graphics functions permit applications to take advantage of hardware features. No longer do programs need to know what hardware can do what or what graphic resolution they can get on a particular device. Using these translation features and standard graphic routines, complex pictures and charts become easy to program.

Clipboard

Quite often in a multitasking environment it is desirable to have applications exchange data. Presentation Manager programs have two such facilities available. The first one is called the *clipboard*.

The clipboard is a virtual area within the system that serves as a scratch work area where applications may communicate data. There are three basic operations that programs may execute on displayed data:

- Cut
- Copy
- Paste

These functions may be performed only on data that is being displayed. The “cut” operation places a chunk of data from the application’s window in the clipboard, leaving a “hole” in the display where the data was.

The “copy” operation copies a chunk of data from the application’s window to the clipboard, leaving the original display intact.

“Paste” inserts the data from the clipboard in the desired position in a target window.

Each of these functions requires a specific action from the user. Applications using the clipboard must supply the user with the facilities to execute these functions. They are not automatic. Both the source and target programs must be written to work with the clipboard if these functions are to be used. The clipboard is not a place in memory at all. It uses the facilities of the application’s memory with shared memory segments. The clipboard simply uses APIs to manage the access to this shared memory.

A user might want this type of exchange capability to use a spreadsheet in one window and a word processor in another. If both are programmed to use the clipboard, the user may incorporate a piece of the spreadsheet into a report in the word processor.

The Dynamic Data Exchange (DDE) functions provide another way applications may exchange data within the Presentation Manager session.

Dynamic Data Exchange

Presentation Manager programs are encouraged to talk to each other and to work together. The Presentation Manager’s powerful set of tools provides the means to exchange data. The clipboard acts as a holding area for a set of data to be passed from one program to another. This method requires an action by the user (the instruction to cut, copy, or paste). Another architected method for data exchange between programs on a real-time basis requires less user action (or none, depending on the implementation). This method is known as *dynamic data exchange (DDE)*.

DDE is both a set of APIs and a protocol; it allows programs to exchange data dynamically as an application is running. DDE requires no special action by the user. It employs a special set of messages similar to those used in Presentation Manager communication. A dialog is created in which applications exchange messages containing data requests, data packets, acknowledgments, and so on.

DDE is more appropriate than the clipboard if the situation calls for a spreadsheet application in one window exchanging data with a graphing program in another. Using DDE, the spreadsheet can send data to the graphing program so that the data being entered may be graphically displayed. The actual power of DDE becomes apparent when the spreadsheet is updated: The graph is modified in real-time as the data in the spreadsheet changes.

Not only does this work with just two programs, but many applications may be involved in a single DDE dialog. The DDE function is invaluable in data collection and analysis programs. No longer is it necessary to collect the data, write it to a file, read that file, and then generate a graph. Using DDE, all this can be accomplished without the intermediate steps. One program simply takes its data and passes it on to another.

The DDE protocol is relatively simple. An application that starts a DDE dialog issues a call to `WinDdeInitiate`. This generates a message to be broadcast to all main windows in the system. The message states where it originated and names the "topic" the originator would like to discuss. Once the application(s) wishing to talk responds, the originator issues a request message to the server. The server sends data messages in response. When the client no longer wishes to receive data, it notifies the server to terminate the dialog. The sample conversation in Figure 6.4 demonstrates this type of communication.

Using DDE, mutually cooperating applications may share information and further enhance the presentation of data. In addition to applications developed to work together, applications that know nothing about each other's existence may also share data, because the DDE protocol is generic. As long as both wish to talk about the same topic, they do not need to know details about the other's operation, only that they wish to talk about a common subject. A graphing application need not know details about the program providing the data to be graphed; it needs to know only the format of the data. It could be a spreadsheet or an analog-to-digital conversion program that is collecting data from test equipment. As long as the DDE initialize request is responded to by the graphing program, the data exchange may take place.

User Shell Program Interaction

Program interaction with the shell is straightforward. These APIs permit programs to install themselves into the start programs list, to add groups to this list, to query information in the task manager, and to send instructions to the task manager. The `OS2.INI` file can be used to store information about an application from one invocation to the next. Access to the `OS2.INI` is through a set of shell APIs. The shell APIs all begin with `Prf` (for example, `PrfAddProgram`).

The shell APIs are used when applications need to interact with some of the shell's data areas. APIs help applications obtain information about running programs to determine the state of the system before taking certain actions.

Summary

The Presentation Manager interface is a very powerful presentation tool. It frees applications from concern about the configuration of the hardware. The Presentation

Figure 6.4

Dynamic Data Exchange Messaging Protocol

REQUESTER

RESPONDER

Does anyone have data
for me?

WM_DDE_INITIATE ==>

<== WM_DDE_INITIATEACK

Yes. I have some.

Send me data.

WM_DDE_REQUEST ==>

<===== WM_DDE_DATA

Here it is.

Tell me if it changes.

WM_DDE_ADVISE ==>

<===== WM_DDE_ACK

OK

<===== WM_DDE_DATA

It changed. Here's the
new data.

<===== WM_DDE_DATA

It changed again!

You can stop now.

WM_DDE_UNADVISE ==>

<===== WM_DDE_ACK

OK

I'm finished talking.

WM_DDE_TERMINATE ==>

<===== WM_DDE_TERMINATE

OK

Manager provides power to present data in many different ways with a minimum amount of work and provides tools to communicate data between applications. It features a variety of new, easy-to-use, and easy-to-program ways of interacting with the user.

This chapter offered an overview of what Presentation Manager programs can do and how they may go about it. The remainder of this section covers the tools and methods required to build Presentation Manager programs.

Building an Application

This chapter explores the procedures and tools necessary to build a Presentation Manager application. A Presentation Manager application is composed of two basic parts. The first is the executable program in which all the instructions that manipulate program objects are stored. The second is the set of resources to be manipulated by the program, including such elements as menus, icons, and dialogs. These resources are generated by the toolkit editors and built into the executable program with the resource compiler.

Source Code Preparation

In OS/2 Standard Edition, the recommended language for programming Presentation Manager applications is C. OS/2 Version 1.2 introduces language bindings for COBOL and Fortran as well, but C is still the preferred language because it is well suited to the control structures of Presentation Manager programs. C is readable, yet powerful and flexible.

The Presentation Manager shell has several unique requirements that must be met by the code generated by a compiler. As of this writing, only two compilers satisfy those requirements: IBM C/2 Version 1.1™ and Microsoft C Version 5.1™. Throughout the remainder of this book, a basic knowledge of the C language is assumed. Only basic language constructs are necessary to build Presentation Manager programs.

INCLUDE Files

With the large number of APIs available and the need for function prototypes, INCLUDE files have become a programming way of life under the OS/2 system. The term *INCLUDE file* is a misnomer; the files in a C program are really header files, because they contain headers for the functions. In many instances, these two terms have been used synonymously, and both names are used interchangeably throughout the rest of the book.

The INCLUDE files are part of the OS/2 Programmer's Toolkit. Each file contains a subset of the headers. All the files combined compose the entire OS/2 API. In addition, the INCLUDE files contain symbol definitions, data types, and structures to facilitate the use of the APIs. Each file covers a specific portion of the calls. For example, PMWIN.H contains all of the APIs, structures, and so on, for all window management (Win calls); PMGPI.H has the Gpi calls, and so on.

The INCLUDE files can be confusing because of the quantity of information they contain. Simplification of the files is provided by the use of conditional sections. By defining symbols in the source code, certain sections of the INCLUDE files can be selected. This enables the programmer to tailor which subsets of the headers are included. Including only the items that are needed is important, because each header or structure takes time to be "compiled" and takes up storage during the compilation.

A set of symbols that selects certain subsections is documented at the top of each INCLUDE file. Each symbol begins with INCL_, and each corresponds to a different section of headers and structures. You need to specify only one file in the source code: OS2.H. The programmer simply determines which symbols to define for the items needed and then includes the file OS2.H.

OS2.H is the main control file for all of the OS/2 headers. Depending on what symbols have been previously defined, it selectively includes other files, which in turn define more symbols and include more files corresponding to the original selections. This cascading action continues through all of the available files. So, all the programmer needs to do is to define the symbols needed and then include OS2.H, for example,

```
#define INCL_INPUT
#define INCL_GPIBITMAPS
#define INCL_WINDIALOGS
#include <os2.h>
```

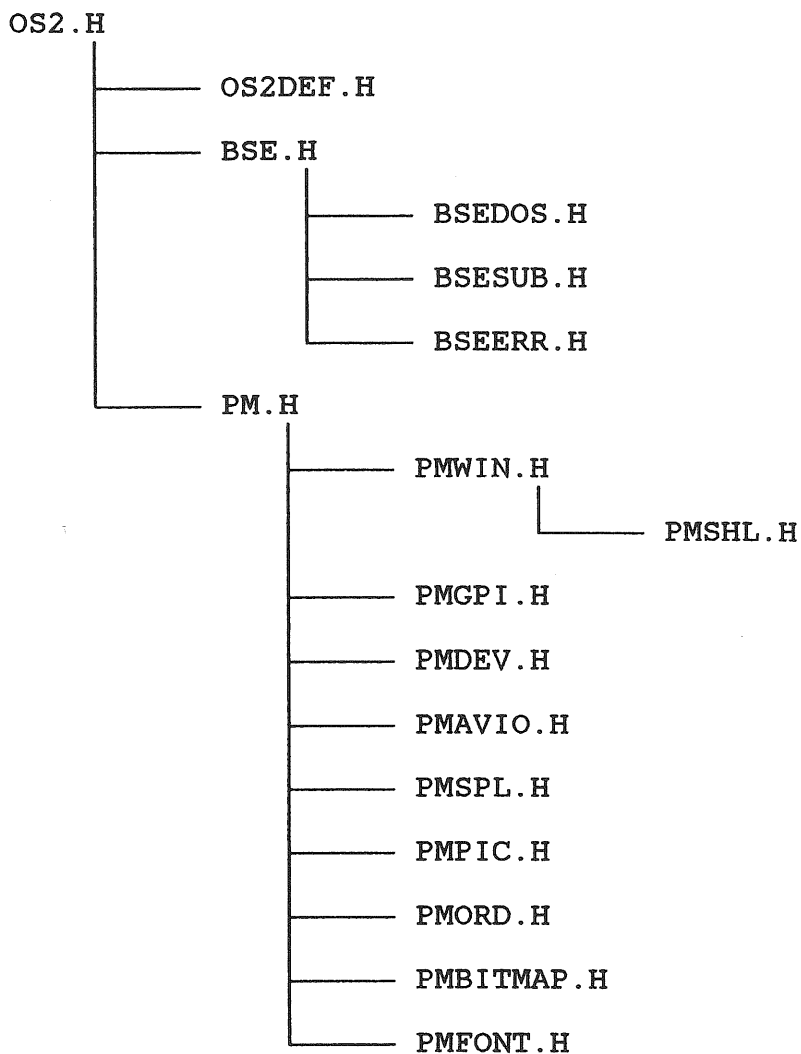
These symbols cause OS2.H and its descendants in the include hierarchy to select only the desired subsections of the API. Appendix E, "INCLUDE File Conditional Selection," contains a reference list of these symbols and what they represent. Figure 7.1 shows the hierarchy of the INCLUDE files.

Naming Conventions

Standards for names and types make code easier to understand and maintain. This chapter discusses some of the naming conventions implemented in the OS/2 Programmer's Toolkit and how they can make life easier.

Figure 7.1

OS/2 Header File Hierarchy



It is important to follow naming conventions as much as possible because they make understanding the APIs easier, and they provide source code consistency both among programmers and among your programs and the API syntax.

Variable naming has always been a challenge, because what is meaningful to one person is not always understood by another. A basic set of conventions helps. To simplify matters, a prefix is added to all variable names to indicate data type and/or function. Here is a list of some of the most common prefixes and what they represent.

Prefix	Meaning
a	Array
b	Byte
c	Counter
ch	Character
dc	Device Context
f	Boolean Flag
h	Handle
i	Index
l	Long
mq	Message Queue
np	Near Pointer
p	Pointer
ps	Presentation Space
rc	Rectangle
sz	Null Terminated String
us	Unsigned Short
v	Global Variable

All of the C language INCLUDE (.H) files use these conventions in their type definitions. Some examples in practice are

hwndMyWindow	window handle for MyWindow
pszTitle	pointer to the null terminated string Title
hmqMsgQueue	handle to the message queue MsgQueue
hdcOutDevice	handle for the device context OutDevice
rcClient	client window rectangle

These conventions are followed throughout this book in discussions of the Presentation Manager programming functions.

Not only is a set of prefixes supplied, but a comprehensive set of types is defined also. Many of the OS/2 APIs require parameters to pass data to and receive data from the routines. Each function has a unique set of items it needs to receive to perform its duties. In prior development environments, there was no set of standards on how data types were to be defined. As long as the number of bytes being passed matched what the function wanted, everything was fine.

The same method works in the OS/2 system as well, but the `INCLUDE` files provide an easy way to make code more consistent and readable. For every data type needed in an API, there is one defined in the `INCLUDE` files to fill the need. These data types are used consistently throughout the APIs and are consistent with the conventions described previously.

The data types used in the APIs are built out of standard “plain language” data types. If every application used its own implementation of the types, porting a program to another system that was not 100 percent compatible would be very difficult. Using standard types, only the header files need change. The structures won’t change; only the low-level component data types need to.

Some of the more common types are listed here. Listing all of the available custom data types in this book would not be feasible because they can change depending on implementations.

Type	Description	
BOOL	Boolean TRUE or FALSE	(2 bytes)
CHAR	Character	(1 byte)
HAB	Anchor block handle	(4 bytes)
HWND	Window handle	(4 bytes)
HMQ	Message queue handle	(4 bytes)
LONG	Signed long integer	(4 bytes)
SHORT	Signed short integer	(2 bytes)
ULONG	Unsigned long integer	(4 bytes)
USHORT	Unsigned short integer	(2 bytes)

Experience suggests that the best way to determine the data types and structures necessary for a particular call is to look in the header file for the function prototype needed. In this prototype is the list of parameters for the API with the data types it expects to receive. A text editor can then be used to search for the type or structure

definition to see exactly what is expected of the calling program. Many of the types and structures are common to many functions. To make your search easier, a list of all the OS/2 function prototypes may be found in Appendix A, "APIs." The INCLUDE files that contain the most commonly used types of functions are listed here.

INCLUDE File	Description
BSEDOS.H	Dos APIs
BSEERR.H	Error code constant definitions for all base OS/2 APIs
BSESUB.H	Vio, Kbd, and Mou APIs
PMAVIO.H	Presentation Manager Advanced Vio APIs
PMGPI.H	Gpi APIs
PMSHL.H	Shell APIs
PMWIN.H	Win APIs except the Shell APIs

In addition to these files, many of the most commonly used data types are contained in the file OS2DEF.H.

Compiling and Linking

The build process usually includes compilation and linking or binding. A Presentation Manager application's build process is similar to that of a traditional program, with a few additions.

When a C compiler is invoked, several parameters (called *switches*) are available. They control different aspects of the compilation, such as whether to optimize the generated code, what memory model to use, and other attributes the programmer would like the generated object code to have.

Presentation Manager programs require two switches to be specified: */Astring* and */Gs*.

/Astring creates a customized memory model. *string* consists of three fields you can specify in any order. One field defines the type of code pointers to be produced:

- s for near pointers and addresses
- l for far pointers and addresses

A second field specifies the size of data pointers:

- n means use near pointers and addresses

- f means use far pointers and addresses
- h means use huge pointers and addresses

The remaining field must contain u. This causes the DS register to be reloaded upon entry to all window procedures.

/Gs turns off stack checking.

An additional switch is required if a program uses Gpi calls. /Zp must be specified to pack structure members.

The following command compiles the program myprog.c with windowing enabled, near code pointers, near data pointers, stack checking off, and packed structure members:

```
cl /c /Ausn /Gs /Zp myprog.c
```

For a more comprehensive list of available compiler switches, consult your C compiler reference material.

The object (.OBJ) file generated by the compiler is not executable. The object module contains information about unresolved external references. The object module must be linked with object libraries to produce an executable load module without unresolved external references.

The OS/2 linker combines the object files with object libraries (.LIB files) to produce executable (.EXE) modules. The object libraries contain either the code for the externally referenced routines (static linking) or reference records that point to the routines residing in dynamic link libraries (dynamic linking).

Memory Models

A *memory model* tells the compiler how to generate pointers to the data segment(s) within a program and allows the compiler to take advantage of large data and code objects. Several memory models are available for the C programmer. Most code generated should run as a small model program unless there is a specific need to use another model. For more details on memory models, you should consult a C language reference.

Module Definition File

In addition to the .OBJ files and the libraries, the linker uses a type of file called the *module definition file* (.DEF). This file contains the attributes that are to be included in the executable file. In the sample .DEF file in Figure 7.2, the module has the WINDOWAPI attribute, which tells the loader that this is a Presentation Manager program. Other choices might be WINDOWCOMPAT, which makes it a VIO-windowable program, or NOTWINDOWCOMPAT, for a full-screen program.

The .DEF file also contains a STUB statement that directs the linker to place the module OS2STUB.EXE in the executable file. When an attempt is made to run the

Figure 7.2

Example Module Definition File

```

NAME      DEMO   WINDOWAPI

DESCRIPTION 'OS/2 Sample Application'

STUB      'OS2STUB.EXE'

CODE      MOVEABLE
DATA      MOVEABLE MULTIPLE

HEAPSIZE  1024
STACKSIZE 4096

EXPORTS
    DemoWindowProc @1

```

program in the wrong mode, the stub notifies the user. For example, an attempt to execute a Presentation Manager program in the DOS box results in a message from the stub saying that the program cannot be run in DOS mode.

The .DEF file also specifies the size of the heap and stack that the program has. There are system defaults for these values, but programs with more windows (and hence more window procedures) and more threads need more stack and heap space. The .DEF file enables the programmer to allocate as much space as needed.

In addition, the .DEF file defines the import and export entry points for the program. These definitions allow a program to call entry points in other modules and enable its entry points to be called by other programs. All window procedures need to be defined as exports, even if no other application programs intend to call them, because the Presentation Manager must be able to make a call to an application's window procedure. All window procedures must be exported or a program will not execute properly.

Using these definitions, the linker resolves external references, builds export entry points, and sets up the stub and header for the program.

The .EXE header contains the information needed by the loader to place the file in memory and transfer control to the beginning of the instruction code within it. Among other things, the header includes the type of program (Windowed, VIO, Full Screen, or DOS). This information tells OS/2 how to run the program. The module definition file tells the linker what information to put into the header.

So far, nothing you have seen about the build process differs from that used for DOS programs, except that the .DEF file is not used in DOS. The differences between the application types appear in the next step.

Resource Compiler

The Resource Compiler (RC.EXE) collects the resource files generated by the toolkit editors (Icon, Font, and Dialog), along with the resource definition file that contains menus, string, and accelerator tables, then constructs the RES file that is analogous to the .OBJ file. In its second pass, the Resource Compiler acts as a linker, combining the .EXE file with the RES file to produce a complete Presentation Manager application. The Resource Compiler may be run in two separate steps or these steps may be performed all at once.

To run the Resource Compiler in two steps, the syntax is

```
rc -r resfile.rc
```

This runs only the first pass of the compiler, which does nothing more than generate RESFILE.RES from the source code RESFILE.RC file. This file then must be incorporated into the .EXE to complete the application.

The second pass of the compiler is invoked as

```
rc resfile.res myprog.exe
```

This pass of the resource compiler binds the RES to the .EXE and completes the build process. The program is now ready to run.

To run the Resource Compiler for both phases at once, generating the RES file and binding it to the .EXE without a second invocation, the command is similar to the second pass just shown, with one minor modification:

```
rc resfile.rc myprog.exe
```

The two-step method is valuable when you have large source code files that have changed, even though the resource program has not changed. The resource files in such a case do not need to be recompiled; they can just be bound to the executable file using only the second pass of the RC.

MAKE Utility

In the last several years, building applications has become a more complicated process. Programs have grown in size and features right along with their systems' capacities. Accordingly, managing the construction of a program has become more of a chore.

Listing

An excellent utility that has emerged is MAKE. There are many implementations of MAKE, but its underlying purpose in all its flavors is to rebuild only the parts of an application that have changed since the last build. MAKE takes as input a file that contains a list of files the target executable(s) is/are dependent on and a list of instructions on how to build them. This is usually called a *makefile*. Using the rules and values in the files, MAKE rebuilds only the portions of an application that need rebuilding. All others are left alone, because their components have not changed.

When you invoke MAKE, it looks at each dependency in the makefile, looks at the dates and times of each of those files, and compares them to the dates of the targets. If a target is dependent on a file that has changed since the date on the target file, that target is rebuilt according to the rules supplied in the makefile on how to build it. Only those portions of the target that need rebuilding are built. For example, if an executable file is dependent on three different .OBJ files and only one of the C files that make up the .OBJs has changed, only that one .OBJ is rebuilt and the executable file (or "executable") is relinked. If the date on the target file(s) is later than the dates of all the files it is dependent on, the target is judged to be up to date and is not rebuilt. The two C compilers previously mentioned come with MAKE utilities that are similar in function. The explanation on how MAKE works becomes clearer through a sample makefile.

Listing 7.1

```
#=====
#
#   Demonstration Application Make File
#
#=====
#
# "all" is the main target, the main file to build is DEMO.EXE

all: demo.exe

# The .RES file is dependent on the resource definition and application
# INCLUDE file DEMO.RC and DEMO.H, respectively
# The command to build the target DEMO.RES is specified directly under the
# dependencies

demo.res: demo.rc demo.h
        rc -r demo.rc

# The target DEMO.OBJ is dependent on the source file DEMO.C and the INCLUDE
# file DEMO.H. The compile command immediately follows this

demo.obj: demo.c demo.h
        cl /c /Asnu /Gs /Zp demo.c

# The main target, DEMO.EXE, is dependent on the object files, DEMO.OBJ and
# DEMO.RES, and the linker module definition file, DEMO.DEF. Once these
# dependent files are there (from the build above), the command to link them
# with the libraries SLIBCE.LIB and OS2.LIB is executed. This creates the
```



```
# .EXE file that is immediately combined with the RES file to produce the final
# Presentation Manager executable.
```

```
demo.exe: demo.obj demo.res demo.def
    link demo.obj, demo, demo, s1ibce os2, demo.def;
    rc demo.res demo.exe
```

```
#####      End of MAKEFILE
```

In this example, the makefile says, "I want to build DEMO.EXE. If the dates of DEMO.RES, DEMO.OBJ, or DEMO.DEF (or any of the files these are dependent on) are more recent than my date, rebuild components that have changed since the last time DEMO.EXE was built." MAKE proceeds to where it needs to use DEMO.OBJ. If MAKE determines that DEMO.OBJ needs to be rebuilt, MAKE uses its rules and dependencies to build the .OBJ file. The same is true for DEMO.RES.

For example, if only DEMO.DEF has changed, MAKE only executes the link and RC steps. No other parts of the application are rebuilt, because MAKE looks at the dates of the other dependencies (.OBJ and .RES) and finds that those parts are already current. MAKE *only rebuilds the parts of an application that have changed since the last build* unless it is specifically told otherwise.

This utility helps to reduce compile and link times. It also helps to make program maintenance easier than with plain batch (or command) files.

Summary

C is the recommended language for writing Presentation Manager programs. Its structures and programming techniques lend themselves well to those of the Presentation Manager. The INCLUDE files supplied with the OS/2 Programmers Toolkit contain all of the function prototypes and data structures needed to write applications. Using the INCL_ definitions along with the main file OS2.H, you can easily include specific subsets of the API.

In addition to the INCLUDE files, the tools provided, such as the Resource Compiler (RC.EXE), enable you to bind the resources to your code.

Compared with DOS, the OS/2 development environment is very structured. There are standards for calling system functions, naming variables, and declaring data types. These standards should not be viewed as restrictions on a programmer's freedom. OS/2 is a very complex system, and these standards make application development easier to understand and code easier to read. Also, by adhering to these standards, programs may be ported to other systems more easily than with previous operating systems in which each programmer (and program) defined different data types, function calls, and so on.

Message Architecture

The Presentation Manager architecture is very complex. It must manage many tasks that run at once, as well as all the resources they require. The display screen is an excellent example. In PC DOS and OS/2 Version 1.0, only one task or process may access the screen at a time. Beginning with OS/2 Version 1.1 and the Presentation Manager, many applications may access the screen at once. Presentation Manager windows demonstrate this concept. At any time, there may be many windows on the screen; they must coexist peacefully. An update to one window must not destroy the contents of any other window.

Another example is the keyboard. Keyboard input intended for one window or application must get to that application and not to any other. The Presentation Manager must juggle all of these resources so that activities occur consistently. A user would not want to enter data on the keyboard expecting to see it on a spreadsheet, only to find that the data went into the word processor's window. Therefore, the Presentation Manager mechanism is needed to manage all applications and keep them informed of the events going on around them.

The Presentation Manager system is event driven. Each event, such as a keystroke or a movement of the mouse, that occurs in the system causes a series of messages to be sent to applications. A *message* is defined as a packet of data containing information about events addressed to the affected areas. The information in the packets gives applications the details about the events.

These messages perform vital functions. They inform applications when events affect them in some way. For example, if a window has been uncovered by the user as another window is moved, the uncovered window needs to get a message telling it that

it needs to repaint the area that was uncovered. When a user requests that a window be closed, a message is sent to that window to let it know, so the application has the chance to save any information it needs before it shuts down.

Messages are the backbone of the Presentation Manager mechanism. They enable applications to talk to the system and to each other. Application input/output is managed by sending messages to them, and the Presentation Manager mechanism acts as the mail carrier for messages sent by them. The paths taken by messages through the system are complicated, because messages may originate from many sources and have a variety of destinations. Each message is unique in why, where, and to whom it is sent.

The function of Presentation Manager messages is communication, which assumes any of three types:

- Application communication with the Presentation Manager
- Communication within an application
- Communication between applications

The Presentation Manager uses a wide variety of messages, and a single message may have different meanings for different applications at any given moment. This chapter examines the structure of a message, the path that messages take through the system, and some programming considerations when you deal with messages.

Message Structure

All messages have the same basic structure. Each one is the same size and contains the same data types. What makes each message unique is its contents. Each message contains the handle of the addressee, the message identifier, and the data to give the recipient details about what action may be needed.

The general components of a message are

- *Window handle*, which identifies the window, or the addressee, to which the message is being sent. All parts of the system communicate with windows by way of their window handles. The messages do not really go to the window, but to the *message queue* or window procedure for that window.
- *A message identifier*, which is a unique numeric value to identify the message. The recipient of the message takes different actions depending on this value. To enhance readability and make messages easier to use, each message is assigned a mnemonic name, so a program looks for a `WM_CREATE` rather than some numeric value. This is true for all Presentation Manager messages. Each mnemonic was designed to indicate the function of the message it represents. When you look at source code, it is much easier to understand what an application is

doing when it receives a `WM_PAINT` rather than looking up the meaning of message number `0x0023`. Each message name is composed of two characters—an underscore and the identifier.

- Two *message parameters*, which are the pieces of data that tell the message's recipient the information needed to process the message. The content of these parameters differs for every message. For example, the parameters for the `WM_BUTTON1DOWN` message indicates that button 1 on the mouse was clicked, what window the pointer was pointing to when the click occurred, and details about where the mouse was in window coordinates and its status. These details are important, because applications need to know not only that an event has occurred but also the details about the event as it affects them.

Threads and Queues

When Presentation Manager messages are sent, they must have a destination. Every application must have a “mailbox” for the messages sent to it. Messages are stored there until the application is ready to process them. The mailbox for messages is the message queue.

A *queue* is a data structure in which data items are added and removed in a specific order. With message queues, items are added to and removed from opposite ends similar to the way a line at a bank is serviced. Messages are added at one end of the queue and they wait in line to be removed at the other end. When messages are sent to an application, they are placed in its message queue. The application then removes the messages and handles each one sequentially.

Before you proceed further, it is necessary to understand some terms that are used extensively throughout the rest of this book. The first is the concept of threads within an application.

Every application has at least one thread of execution. A *thread* is the smallest unit of execution in OS/2. A single application may have many threads concurrently executing to perform its different tasks. Every thread in a Presentation Manager program that causes messages to be sent must have its own message queue. Because it is sometimes difficult to know whether a function that your thread is calling may be using messages, the safest thing to do is to create a message queue in each thread that makes Presentation Manager function calls. The reason is that when a message is sent, replies may be returned as well as messages initiated by the receiver. Every message is important, and any one that is not handled can cause the system to act unpredictably. If a thread that is to receive a message does not have a message queue, that message is lost. This important fact must be kept in mind when you develop applications.

Another concept that must be explained is the idea of “handling” a message. Each application has its own main message queue. Through this queue the system communicates with the application. If messages are not handled expediently, they back up in the queue and slow down the execution of an application. Therefore, all messages should be “handled” within 1/10 second to ensure that they are acted upon in a timely fashion. So handling a message means that the application has executed the proper instructions to begin processing the message and has returned to get the next message. It is quite possible to fully process a message well within the recommended time. If some lengthy processing must be done, the application should start a thread to complete the processing while allowing the main thread to return to continue processing.

An exception to this would be a message requiring processing such that the application cannot continue until that processing has been completed. In this case, the system provides facilities for the application to inform the user. A prime example is a program that is reading or writing a data file. In many cases, the program cannot continue until the file is read or written. In this case, the user should not be able to interact with the program until the I/O operation is complete. The program can be written to change the system pointer to the hourglass icon to indicate to the user that he or she may not continue until it is released.

One warning here: The program must contain a contingency routine in case the I/O operation encounters an error. If this happens and the pointer is captured and changed, no other program interaction can take place. From the multitasking standpoint, no application should determine that it is more important than any other application in the system. If such prioritizing is allowed, an application can monopolize the system and be “ill behaved.” Applications that do not peacefully coexist with others in the system are not popular.

The other concept to understand here is *posting* versus *sending* a message. The next section discusses this subject in depth, but we introduce the terms here.

When a message is posted, it is placed directly in the application’s message queue. The thread that does the posting simply places the message in the queue and continues its execution asynchronously. It does not wait for the message to be handled.

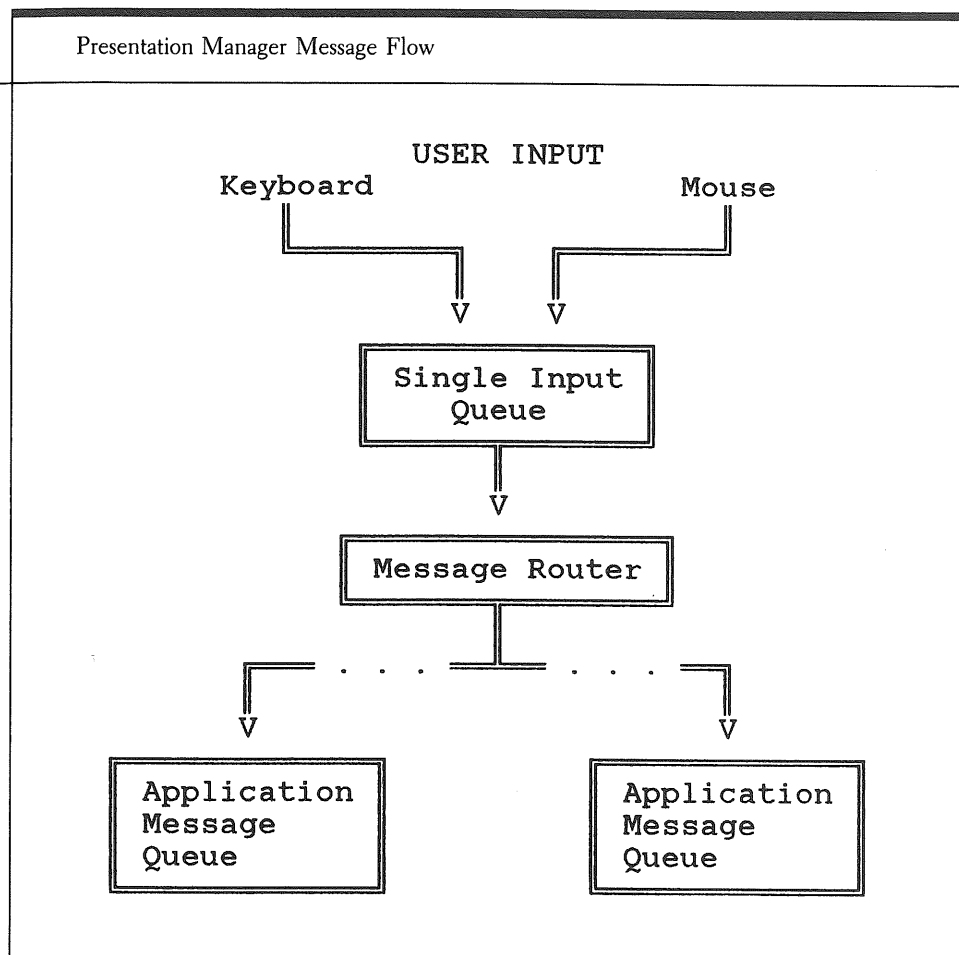
Sending, on the other hand, is a synchronous action. The sender of a message does not really place a message on the queue. The sender is actually making a call to the window procedure of the application it is “sending” the message to.

Chapter 12, “Window Messages,” explores the differences between sending and posting messages in depth.

Message Flow

Figure 8.1 shows the message flow within the Presentation Manager session. Messages originate from two sources: user input and applications. User input messages originate

Figure 8.1



from keyboard or mouse interaction. Application messages are either sent or posted by an application thread in the system. This may be a thread of the application receiving the message or another thread in the system.

Messages generated by an application thread, whether it be the same thread as the recipient of the message or some other thread, are either sent or posted to the recipient queue by way of `WinSendMsg` or `WinPostMsg`, respectively. Both APIs communicate a message to the application, but their process controls are different. The `WinSendMsg` function is a synchronous send, whereas `WinPostMsg` is asynchronous. As described earlier, when a thread sends a message, it does not continue until the recipient of the message has handled it and returned. When a message is posted, however, the thread that posts the message continues without waiting for a response.

Messages generated by the keyboard or mouse go into the single input queue. The single input queue receives all user-initiated input (keyboard and mouse) that enters the system. Any keystroke adds an element to the queue, as do all mouse movements and button clicks. Each message has its own parameters to tell the system what the details are. For example, a keyboard message contains the raw scan code of the key or key combination that was pressed.

The single input queue is filled and emptied on a first-in, first-out (FIFO) basis. Messages may fill the queue if they come in faster than the system can process them. In this case, the system empties the queue in the order the messages came in until the queue is empty. This offers the user a large type ahead (or “click ahead”) capability.

Once the message leaves the single input queue, it goes to the message router. This logical “black box” performs several functions. It is responsible for removing messages from the queue. Its job is to decide which thread is to receive the resulting message, translate the raw data values into Presentation Manager messages, and send the message to the right place. For keyboard input, it queries the system to determine which window has the input focus. It sends the `WM_CHAR` messages to that window. For mouse messages, the router sends the messages to the window that is currently below the pointer on the screen.

Keyboard messages contain the scan code of the key or key combination that was pressed. In this instance, the message router translates that code into a combination of a virtual key code, shift state, and status flags, such as whether the Ctrl key was pressed. In most cases, the raw scan code is also sent in the translated message. Now the router does the keyboard accelerator testing to see if the key or key combination pressed is an accelerator defined to the system. Such an example is the Alt-Esc key combination.

When an accelerator is encountered, the message router translates the accelerator into a `WM_COMMAND` message to be treated separately by the program. The `WM_COMMAND` message contains information that identifies the accelerator and lets the application know which one has been activated.

When a system accelerator is entered (such as Alt-Esc), the message router causes the system to take immediate action on it and not even pass it to the program. Such system accelerators are part of OS/2 and cannot be intercepted by Presentation Manager programs.

Mouse messages are “hit tested” against the window currently under the pointer on the screen at the time the message is generated. The message router sends a `WM_HITTEST` message to that window. The window responds in one of several ways:

1. It indicates that the message should be discarded.
2. It indicates that it does not want to handle the message. In this case, the window below it in the z-order gets the message as if it were sent directly there. This action may continue down through the bottom of the z-order.

3. It indicates that it will process the message.

Mouse input may not always go to the window below the pointer. An application may want to capture mouse messages anywhere on the desktop for a certain time. In this instance, the application may issue a `WinSetCapture` to capture all mouse messages. In this case, all mouse messages go to the application that issues the `WinSetCapture` until a `WinReleaseCapture` is done. This might be done, for example, if a window wishes to track mouse movement outside its own window.

Once the message router finishes processing the data, the message is then sent to the application thread's message queue for processing by the application. The information is passed in the form of `WM_*` messages. These are the messages generated by the message router, as well as the messages that may be sent or posted from another application thread. Keystroke data described previously is sent to an application as a `WM_CHAR` message. With an accelerator, it will be a `WM_COMMAND` message. Mouse messages are received at the application level as `WM_MOUSEMOVE`, `WM_BUTTON1DOWN`, and so on. Individual messages and their processing are discussed fully in Chapter 12, "Window Messages."

Once these messages are in the application's message queue, the application must call the `WinGetMsg` API (or in certain cases, `WinPeekMsg`) to remove them for processing. An application's main thread is a message input thread. Its primary job is to sit in a loop, processing messages for the application. *Processing* in this context means simply removing the message from the message queue and dispatching it (via `WinDispatchMsg`) to the window procedure.

The `WinDispatchMsg` function takes the item that was removed from the queue and breaks it out into its parts, those being the items identified as the parts of a message. This is a `QMSG` structure that was built from the message components by the `WinPostMsg` API. (As you have seen, `WinSendMsg` does not place a message in the queue.) `WinDispatchMsg` breaks the `QMSG` structure into its components again. Once they are separated, `WinDispatchMsg` performs its other task—to make a far call to the window procedure of the window associated with the handle specified in the message.

The parameters of the window procedure are the message components just described. This is all accomplished on the same thread of execution. This explains why the 1/10 second rule is needed. The program cannot get the next message from the queue until the window procedure has returned control to it. (See Figure 9.1.)

Message Priority

As stated earlier, user input (via mouse or keyboard) is processed synchronously. Items entering the single input queue leave in the same order. For an application queue, however, messages are prioritized. The general priority order is

1. WM_SEM1
2. Posted messages
3. User input messages (generated by the message router)
4. WM_SEM2
5. WM_TIMER messages
6. WM_SEM3
7. WM_PAINT messages
8. WM_SEM4

WM_PAINT is next to last in the list because the visual update is least important as far as time-critical response goes. Data integrity comes first; the painting will be done soon enough, but the data must be consistent.

Only one thread may actively process user input at a single moment in time, and it must be the thread with the input focus or mouse capture. This does not prevent other threads from processing other types of messages, however. For example, an application that displays a window with nothing more than a clock that is updated every second does not need user input to run. The user may type data into a spreadsheet program in one window that contains the focus, while the clock program is responding to WM_TIMER messages for its window and generating WM_PAINT messages for itself. The clock's window will be updated asynchronously to the spreadsheet window as it does not need user input.

To further illustrate this example, a graph program running in another window and using Dynamic Data Exchange may be responding to changes in the spreadsheet by updating its currently displayed graph. Like the clock window, the graph program does not have the focus but responds to these nonuser-input messages.

Anything happening in the system that causes messages to be sent to windows may be processed immediately by those windows. They do not have to wait for user interaction. As in OS/2 Version 1.0, the user may interact with only one program at a time. That program may be communicating with another, but only one is visible at one time. Under the Presentation Manager shell, dynamic changes in many programs' windows can be seen simultaneously, even though the user can only interact with one at a time. Additionally, because the Presentation Manager shell handles what to tell windows when they are moved, uncovered, and so on, all of these changes happen without explicit action by the user. The messages inform the applications what is happening even if the user is not working directly with that window.

Summary

The Presentation Manager architecture is what enables OS/2 to allow many applications to access the different physical devices in a computer system simultaneously. OS/2 provides isolation among sessions, but the Presentation Manager session must be managed to enable several programs to share the screen and exist in harmony while they do not know anything about each other.

This event-driven, message-based architecture allows for synchronization of user input and also provides for asynchronous message passing between applications that do not require user input. Applications must process messages in a timely manner to avoid slowing down system response time.

Messages are a way of life in writing Presentation Manager programs, whether they be user-, application-, or system-generated. You will come to see that almost anything can be accomplished in the Presentation Manager system through the use of messages.

Presentation Manager Program Structure

The structure of a Presentation Manager program is similar to any other well-structured program. It contains two primary parts: a main control routine and a window procedure. Both are illustrated in Figure 9.1.

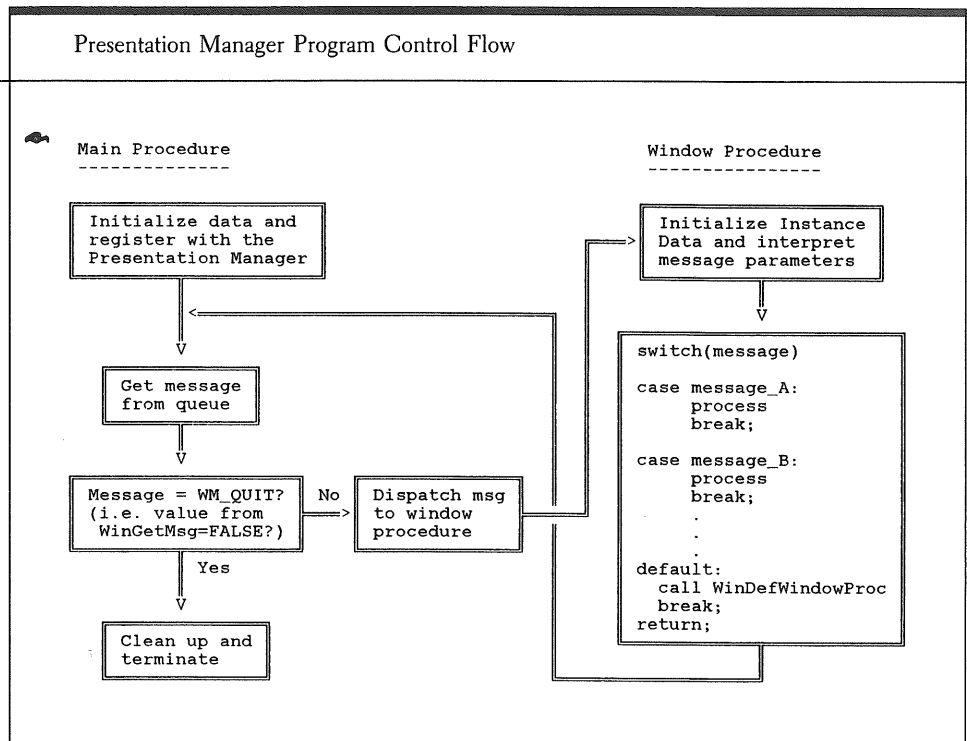
Main Processing Routine

The main routine of a Presentation Manager program is usually not large, yet it performs a huge function. It is responsible for all initialization, input, and termination of the application. This includes handling application-specific data as well as performing vital Presentation Manager housekeeping functions. These functions include initializing queues and allocating structures that the Presentation Manager requires. It also is the thread of the application that receives the messages designated for that application. Once the initialization is done, the main routine must remain in a loop so it is constantly ready to receive messages and to dispatch them to the window procedure.

When an application is started, the first function that gets control is the main processing routine. Now the main processing routine must set up all of the global data areas to be used by the application. It then needs to register itself with the Presentation Manager shell and create the data areas required for Presentation Manager processing. Once this is done, this main control routine goes into a loop known as the “get message loop,” shown in this example:

```
while(WinGetMsg(hAB,                /* As long as we get a non-NULL return */
      (PQMSG)&qmsg,                /* code from WinGetMessage */
      0, 0, 0))
```

Figure 9.1



```

        (HWND)HWND_DESKTOP,
        0,
        0))
    {
        WinDispatchMsg(hAB,
            (PQMSG)&qmsg);    /* Dispatch the message to the correct */
    }                        /* window */

```

What this code fragment is saying is that while the return code from the API `WinGetMsg` is not equal to the value `NULL`, the system should dispatch the message to the window procedure. Behind the scenes, what this is really doing is taking the item out of the queue with the `WinGetMsg` call and placing it in the variable `qmsg`. This item is a message packed together into one structure. The `WinDispatchMsg` call breaks this structure into its components, the pieces that make up the parameters of a window procedure. It is no coincidence that these four parameters exactly match the four items that compose a message. The other job of `WinDispatchMsg` is to send this message to the addressee.

This is all done on the same thread of execution making the 1/10 second rule necessary. The loop cannot continue until the window procedure returns. When the

window procedure returns, the main procedure issues the next `WinGetMsg` to indicate that the previous message has been processed. If this does not happen within the time guideline, messages back up in the queue and cause the application to slow down.

This loop removes messages from the application queue as long as there are messages to be removed. Once the queue is empty, the next call to `WinGetMsg` causes the thread to block. The thread remains blocked until a message comes into the application's queue. At that time, the thread is awakened and processing continues until once again there are no more messages in the queue. This again causes the thread to block, and the process continues throughout the life of the application until the loop is exited.

The loop exits when `NULL` is returned by the `WinGetMsg` call. This only occurs when a `WM_QUIT` message is the message removed. That is the notification to the application that some event has occurred in which the user wants the application to shut down. When the "get message loop" is exited, the main routine must then perform functions to "clean up after itself."

These clean-up functions may include closing open file handles, terminating interaction with I/O device drivers, and storing data the application needs. These functions must include calls to terminate the interaction between the application and the Presentation Manager shell. For example, the windows and message queues must be destroyed and the application must deregister itself with the Presentation Manager. The system performs these operations if the application does not do it for itself, but it is best for the application to handle this work.

Once the housekeeping is done, the main routine calls `DosExit` and terminates.

Window Procedure

The window procedure is the workhorse of a Presentation Manager application. Messages come here when they are dispatched in the main loop or sent by other applications. The basic function of the window procedure is to look for messages the application is interested in and to handle them. The procedure passes all other messages to the default window procedure, `WinDefWindowProc`. `WinDefWindowProc` is discussed in detail in the next section. For now, it is only necessary to know that the system has default processing for every message.

A window procedure is invoked by a far call. It executes the same way as any other function. Every window procedure must have certain parameters in its parameter list. These values tell the window procedure everything it needs to know to handle the message:

- The handle of the window the message is destined for
- The message identifier

- Message Parameter 1
- Message Parameter 2

When you invoke the window procedure, it first initializes any instance data needed to handle messages. Remember that this function is called every time a message is sent to the application. Each message causes another call to the window procedure, so data that needs to be retained across calls to the window procedure must be stored as global data. Other data that is only needed on a per-call basis may be allocated and initialized when the window procedure is called. When the window procedure exits, that instance data is gone.

When the initialization is complete, the window procedure looks at the messages and handles the ones it is interested in. This is usually done by a C language "switch" control structure. The way this works is that the message identifier is the target of the switch. Each case within the switch tests to see whether the target of the switch is equal to that case. If so, execution begins at that point. If not, the next case is tested, and this testing continues until either all cases are exhausted without a match or the "default" keyword is encountered. If no cases are matched and there is a default keyword, execution continues at that point as if it had been a case match. The default processing is to pass the message on to `WinDefWindowProc`. Using this switch, the window procedure looks for and handles all messages it is interested in and passes the rest on for the default processing.

This type of program structure is similar to the traditional structured programming style, in that there is a main routine that takes input and a main worker routine that uses other, smaller workers to carry out functions. A big difference between these types of applications and a Presentation Manager program is that there is default processing for every message in the system. An application only needs to handle messages that have special meaning to that program and let the system handle the rest. This gives applications many built-in functions and leaves the program designer free to concentrate on program functions, delegating the detailed I/O management to the operating system. For example, programs need not concern themselves with what happens when a window is moved unless that event has significance to them.

Another example is keystroke messages. An application only needs to be concerned with whether certain keystrokes have special meaning to it when it is in a particular state. If it is not, the application simply passes the `WM_CHAR` message on and forgets it. The system handles that. If a Ctrl-Backspace was pressed and that key combination has significance to the program, it is processed by its window procedure. If that key combination has no significance to the application, it simply sends it to `WinDefWindowProc`. Only those messages that are of interest to the application are coded in the window procedure.

The next group of chapters discusses how each message has a section in the switch statement and how the final instruction of the window procedure returns to give control back to the main routine so that it may continue its get message loop.

Summary

Structuring a Presentation Manager program is nothing more than writing a modularized program. The window procedure is simply a procedure that accepts a specialized format and a standard set of parameters. The main routine is the classic idea of a program's primary function. It initializes processes, dispatches work to the worker routine(s), and cleans up when all the work is done.

modu-
alized
a of a
rou-

SECTION FOUR

WINDOW PROGRAMMING

Okay, you've read how to use OS/2 Standard Edition Version 1.20 and the Presentation Manager and seen the fundamentals of the system. You have been introduced to some of the tasks this system can perform. Now it's time to sink your teeth into some code and function usage.

In this section, the concepts and functions previously discussed are put to practical use. Code examples are shown and, more importantly, we give reasons why and how things are done the way they are. Using functions in the Presentation Manager session is very simple, thanks to the powerful OS/2 API. The real key to programming under the Presentation Manager shell is to understand how the system works and flows. Once that is accomplished, the functions you wish to program become very straightforward.

Throughout this section, window management functions are discussed. This discussion includes how windows are controlled and how programs work with the OS/2 and Presentation Manager system to manage input and output with windows.

Sam

Presentation Manager Versus Traditional Programming

In many ways, Presentation Manager programs are not much different from well-structured programs written for other operating systems. The program structure is relatively the same. The big departure is that you have to understand and work with the Presentation Manager's message-passing architecture and use an event-driven program design rather than a procedural design.

Throughout the remainder of the book, an understanding of the concepts presented previously is assumed. Terms such as *message queue*, *window procedure*, and *WinDefWindowProc* will become part of your standard vocabulary.

In this chapter, a traditional program that writes the string "Hello World" on the screen in text mode is shown. Then a Presentation Manager version of the same kind of program is presented. Obviously, the Presentation Manager program presents its message in a window.

Sample VioWrtTTY Program

The program in Listing 10.1 is a version of the infamous "Hello World" program almost everyone has seen at one time or another. This version uses the base OS/2 API VioWrtTTY. The function takes a character string and writes it at the current cursor location on the screen.

The program begins by using the include structure mentioned in Chapter 9 to include the OS/2 base headers. The variables needed for the VioWrtTTY function

are then allocated and initialized. Note the use of the naming conventions discussed in Section 3. Using these conventions consistently makes code more readable, especially when the program code starts to grow. Once the data areas are allocated and initialized, `main()` takes control. Inside `main()`, the call to `VioWrtTTY` writes the string to the screen and the program terminates.

Listing 10.1

```

/*****
/*
/*          Sample VioWrtTTY program to say Hello World          */
/*
*****/

#define INCL_BASE          /* Identifier to include OS/2 base functions */
#include <os2.h>            /* Include the master header file      */

USHORT hVioHandle = 0;     /* VIO handle for the screen          */
CHAR   pszHiString[]="Hello World"; /* String to print */

main ()
{
    VioWrtTTY(pszHiString,sizeof(pszHiString),hVioHandle); /* Write the string */
}

```

This program is short and to the point. As you can see, it requires a minimum of code to place data on the display. However, all this does is place the data on the screen in a plain, single-line fashion. The data display methods that you may wish to use, such as painting pictures, graphs, and detailed character functions, are not provided in full-screen sessions. As in previous PC operating systems, you must spend as much time writing the data display functions as you do processing the data.

Full-screen programs have their place, as do all programs. OS/2 full-screen sessions are provided for programs that do not need or wish to use Presentation Manager functions. Next, you will see the same type of “Hello World” program written for the Presentation Manager session.

Sample Presentation Manager Program

As you can see at first glance, the Presentation Manager version of the “Hello World” program in Listing 10.2 is a bit longer than its “traditional” counterpart in Listing 10.1. The functions offered by the Presentation Manager window interface are very powerful. Along with that power is some housekeeping that must be done. Following the code is a line-by-line analysis that explores what each API function does and where each one should be used.

Listing 10.2

```

/*****
/*
/*
/*          Presentation Manager "Hello World" Program
/*
/*
*****/

#define INCL_PM          /* Define to include Presentation Manager headers */
#include <os2.h>          /* Include the master header file */

#define HELLOICON 1      /* Define the icon ID */

HAB      hAB;             /* Handle to an anchor block */
HMQ      hmqHello;        /* Handle to a message queue */
HWND     hwndHello;       /* Window handle */
HWND     hwndHelloFrame;  /* Frame window handle */
POINTL   mypoint;         /* Point structure used to draw text */
CHAR      szClassName[]="HelloClassName"; /* Name for the window class */
CHAR      szMessage[]   = "Hello World"; /* Message for the window */
CHAR      szTitle[]     = "Intro Sample Program"; /* Title Bar string */
ULONG     MyCtlData     = FCF_STANDARD & ^FCF_MENU & ^FCF_ACCELTABLE; /* Frame flags */

/*Function prototype*/
MRESULT FAR PASCAL HelloWndProc( HWND, USHORT, MPARAM, MPARAM );

/*****
/*
/*          Main function
/*
/*
*****/

USHORT cdecl main( )      /* main routine; required in C */
{
    QMSG qmsg;             /* Define the message queue item structure */

    hAB = WinInitialize(NULL); /* WinInitialize initializes the thread */
                                /* and returns a handle to the anchor block */

    hmqHello = WinCreateMsgQueue(hAB, 0); /* Create the message queue and */
                                           /* return a handle to it */

    if (!WinRegisterClass( hAB,
                           (PSZ)szClassName, /* Register the */
                           (PFNWP)HelloWndProc, /* class for the */
                           CS_SYNCPAINT | CS_SIZEREDRAW, /* client */
                           0))
        return((MRESULT)1); /* If the call fails, return */

/*****
/*
/* The following function call creates the window itself. The window is */
/* visible and has an icon, uses the frame control flags defined above, has */

```

PRESENTATION MANAGER VERSUS TRADITIONAL PROGRAMMING

```

/* the class name and title defined above, and returns the Frame window */
/* handle in hwndHelloFrame and the client window handle in hwndHello */
/*
/*****

    hwndHelloFrame = WinCreateStdWindow( HWND_DESKTOP,
                                        WS_VISIBLE | FS_ICON,
                                        &MyCtldata,
                                        (PSZ)szClassName,
                                        (PSZ)szTitle,
                                        OL,
                                        (HMODULE)NULL,
                                        HELLOICON,
                                        (HWND FAR *)&hwndHello );

/* Get the messages and dispatch them */
    while( WinGetMsg( hAB, (PQMSG)&qmsg, (HWND)NULL, 0, 0 ) )
    {
        WinDispatchMsg( hAB, (PQMSG)&qmsg );
    }

    WinDestroyWindow( hwndHelloFrame ); /* Destroy the window when done */
    WinDestroyMsgQueue( hmqHello ); /* Get rid of the message queue */
    WinTerminate( hAB ); /* and deregister with PM */
}

/*****
/*
/* This is the window procedure for the client of the standard window. */
/* It processes window messages of interest to the client. All others will */
/* be passed on to the default window procedure. */
/*
/*****

MRESULT FAR PASCAL HelloWndProc( hWnd, message, lParam1, lParam2 )
HWND hWnd; /* The window handle */
USHORT message; /* The message itself */
MPARAM lParam1; /* Message parameter 1 */
MPARAM lParam2; /* Message parameter 2 */

{
    HPS hPS; /* Handle to presentation space */

    RECTL rect; /* Structure for a rectangle */
    POINTL pt; /* A point structure */
    switch (message) /* Test for messages of interest */
    {

        case WM_CLOSE: /* If message is to close, */
            WinPostMsg( hWnd, WM_QUIT, OL, OL ); /* post a message to my window */
            break; /* handle to quit */
    }
}

```

```

case WM_CHAR:

/*****
/*
/* Character input. The low word of lParam1 contains the key type flags,
/* the high word of lParam1 contains the autorepeat count, and the low word
/* of lParam2 contains the character code.
/*
/*
*****/

    if ( SHORT2FROMMP(lParam2) == VK_F3) /* If the key is F3,
    {
        WinPostMsg(hWnd, WM_CLOSE, 0L, 0L); /* cause termination
        return((MRESULT)TRUE);
    }
    break;

case WM_PAINT:
    hPS = WinBeginPaint( hWnd,
                        (HPS)NULL,
                        (PRECTL)&rect ); /* Get a PS handle

    WinFillRect(hPS,
                (PRECTL)&rect,          /* Fill the rectangle with
                (LONG) CLR_PINK);       /* pink

    pt.x = 70L;                        /* Set up the x and y to
    pt.y = 70L;                        /* draw the string
    GpiCharStringAt( hPS,
                    (PPOINTL)&pt,
                    (LONG)sizeof(szMessage)-1, /* Draw the text
                    (PSZ)szMessage );
    WinEndPaint( hPS );                /* Release the PS
    break;

default: /* If you get here, the message is not of interest */
    return(WinDefWindowProc( hWnd, message, lParam1, lParam2 ));
    break;
}
return((MRESULT)0L);
}

```

The first section of a Presentation Manager program is the main processing routine. This is a function called `main()`, and it is present in all C language programs. When a program is executed, control passes from the operating system to this routine. It is this function's responsibility to control the execution of the program. There may be many functions in a C program, but `main()` must always be present and is the first to get control.

The first element of the main processing routine includes the headers and data types needed by the program. Using the naming and INCLUDE file conventions described in the previous section, along with the complete list in Appendix D, the conditional sections to specify which headers to include are used. For this example, all that were used were the Presentation Manager functions and their related types. This program uses no Dos calls directly, so they do not need to be included. The inclusion of the proper headers and definitions for this program can be accomplished by defining the symbol INCL_PM and then including the master header file, OS2.H. This is a bit of overkill, because INCL_PM causes the inclusion of headers for all the Presentation Manager APIs. Using the list of INCL values listed in Appendix D, the number of headers can be reduced significantly. This step reduces compile time and heap space consumption during compilation.

Once the headers are included, the symbols local to the program need to be defined. In this case, there is only one—the identifier of the icon to be used when the program window is minimized. As you will see further in the program when the .RC file is discussed, this symbol is associated with the icon file. When the user selects the minimize option, the window shrinks and is displayed as the picture in the file associated with this symbol. This is sometimes called “iconizing” the window.

Next, the global data areas are allocated and initialized if necessary. Many variables required by the Presentation Manager system, such as message queues and window handles, are best made global. The reason for this is that quite often there are many functions in an application that require these values. Storing them in global data areas relieves the overhead of querying the system for these values every time they are needed.

The other step before program execution begins is prototyping the window procedure. It must be exported in the module definitions file, because the Presentation Manager functions must be able to call your window procedure, so they must have a way to get addressability to it. This prototype consists of the window procedure name and the data types it expects to receive.

Now the program has begun executing inside main(). Any local variables that are needed are first allocated and initialized, then the executable code begins. The Presentation Manager functions that must be executed in order to work with windows are outlined here.

Call	Function
WinInitialize	Obtain an anchor block handle
WinCreateMsgQueue	Obtain a message queue handle
WinRegisterClass	Identify window procedures
WinCreateStdWindow/ WinCreateWindow	Create a window

WinGetMsg	Remove a message from message queue
WinDispatchMsg	Call the Window procedure
WinDestroyWindow	Destroy the window
WinDestroyMsgQueue	Destroy the message queue
WinTerminage	Release the anchor block

The next step is to register with the Presentation Manager system. This is done with the WinInitialize API. This gives the program a handle to an anchor block (HAB). The system does not use the information in the HAB, but to be portable to other SAA systems that do use it, the HAB must be included in many of the API calls. The WinInitialize call must be the first Presentation Manager API call issued by programs running in the Presentation Manager session.

Next, items that will be used in processing messages need to be created. The delivery box for messages is the message queue. This is created by issuing a call to WinCreateMsgQueue. Two values are passed to this call: the HAB that was returned as a result of the WinInitialize call and the desired queue size. The queue size is specified in a number of messages. The zero used here tells WinCreateMsgQueue to create a queue of the system default size, which is 10 messages. This is usually adequate for most programs.

Every window in the Presentation Manager session has a *class*, which defines the behavior of the window. Each window that belongs to a certain class uses the window procedure associated with that class. A window procedure defines the behavior of all windows belonging to the class the window procedure is associated with. There are several system-defined window classes in OS/2 Version 1.20, each with a unique window procedure. The classes are listed here:

Class Name	Type of Window
WC_COMBOBOX	Combination of listbox and entry field
WC_MENU	Menu control
WC_MLE	Multiline edit control
WC_LISTBOX	Listbox control
WC_FRAME	Frame window
WC_ENTRYFIELD	Text entry field
WC_BUTTON	Button (like the ones used in dialogs)
WC_SCROLLBAR	Scroll Bar

WC_TITLEBAR

Title Bar

WC_STATIC

Static window for output only text

These system-defined classes are also called *public classes*. They are discussed in more depth in Chapter 11.

If you wish to control the behavior of particular windows, you can register your own private window class. This class is private to your program and controls the details of the window behavior as you wish. This is why the API `WinRegisterClass` is used. Public classes are registered by the operating system at system initialization time. These classes are accessible to any program running in the system. The class you create with `WinRegisterClass` is only valid for the process in which it is created. No process may use a window class belonging to another process.

The `WinRegisterClass` call in this example registers a new private class with the class name, `HelloClassName`. The reason for registering a new class is that the system-defined classes do not provide the function desired for this window. The programmer wants the window in this example to be pink and display a single message. This behavior is not built into any of the system-defined classes, so a new one must be created. You will find that the client area of your windows quite often belongs to your own private class and uses other system-defined classes to do other work.

No application-registered classes may be registered with the same name as system-defined classes. To avoid naming conflicts, the same naming convention as the system classes (`WC_*`) has not been used in this example. Along with the class name, you specify the window procedure that defines the behavior of all windows created belonging to this class. This window procedure is defined later in the program. Only the address of the window procedure needs to be passed to `WinRegisterClass`.

Along with the class name and window procedure, each class has certain styles associated with it. Each of the public classes has styles that, in addition to the window procedure, control certain aspects of their behavior. This discussion does not go into them in depth, but for completeness, the styles used here cause the window to be synchronously repainted and redrawn when sized, respectively.

Now everything is set up so that the program can use Presentation Manager services. The next step before the window can be used is to create it. For the demonstration here, a standard window is used. This type of window is created using the API `WinCreateStdWindow`. Values for all the pieces in the window display must be given to the function.

First, `WinCreateStdWindow` is told that this window is a “child” of the desktop window (`HWND_DESKTOP`). The next parameter tells it that the window style should be visible and that the frame style has the `ICON` attribute. This is because the window should be visible when created and have an icon when minimized.

Next, you specify the frame control flags to tell the function what controls it will have. The system has a list of predefined values for all of the public classes of controls

available to applications. These controls include windows such as the Title Bar, system icon, and sizable border.

In this case, all of those except the accelerator table and the menu bar are used. (You were promised this one would be simple!) All of the flags available here are combined to form a single `ULONG` value. They are combined with the OR operator, and flags may be removed from a composite value (such as `FCF_STANDARD`) with the AND operator.

The next two parameters tell the function to which class the client area of the standard window belongs and what string of text to place in its Title Bar. The following two parameters are null values, because they are not needed in this simple program. They describe extra client window styles and a module ID for a program that has resources residing in a DLL module.

The next parameter should look familiar; it is the icon identifier defined earlier. This gives the ID to the frame window for the icon it uses when users minimize it. The final parameter is the handle to the client window of this standard window. The client's handle is returned in this parameter, and the handle to the frame is returned from the function call.

It should be becoming clear by now that the standard window is *not* one window. It is really a complex network of windows owned by the frame. Each piece of the standard window is another window. For example, the client window, Title Bar, and minimize and maximize icons are all windows. Each is owned by the frame except for the client window. By turning on certain bits in the "control flags" parameter in the `WinCreateStdWindow` call, you can control which windows are created as part of your standard window.

Now that you have created the window, the program can start processing messages for it. At this time, the main processing routine is placed into a loop to process messages destined for the window.

The private class previously registered (and its window procedure) defines the behavior for and processes messages destined for the client only. If the application is not concerned with handling sizing, movement, and such, the program can let the system-defined window procedure for the frame handle that. The only concern in this example is the client window.

Next, the `GetMsg` loop is entered. The only required API inside the loop is `WinDispatchMsg`. This API breaks up the `QMSG` structure that has been removed from the queue by `WinGetMsg` into the four parts of a message. It then makes a far call to the window procedure to process the message. When the window procedure returns, it goes back to the top of the loop and starts again.

The main routine stays in this loop until it receives a `WM_QUIT` message. At this time, it breaks out of the loop and cleans up after itself. First, the window that has been created must be destroyed. All that must be done is to destroy the frame. All of the other windows are children of the frame, so when the frame is destroyed, all the

children go away, too. Also, the window cannot be left there without some means of getting messages to it, because such stray messages can cause the system to behave unpredictably.

Once the window is gone, the message queue should be destroyed and `WinTerminate` should be called to terminate the registration with the Presentation Manager system. The program then ends.

These are the basic functions every main routine in a Presentation Manager program must perform. There are many more available functions covered later, but these are the fundamental ones. Now that you have seen how the program has to interact with the Presentation Manager mechanism to start up and close, take a look at how to control the window that has been created.

Sample Program Window Procedure

The window procedure that has been associated with the class that this window belongs to defines the behavior of the client window. It is nothing more than a function that gets called just like any other. The procedure needs to be able to handle any message as it gets called every time a piece of the system wishes to communicate with the window. If it does not want to handle a particular message, it makes a call to `WinDefWindowProc`. This special procedure is designed to respond to any message that a window may receive.

The window procedure must be defined as a "FAR PASCAL" function that returns an `MRESULT` value. All window procedures return an `MRESULT` and must have the PASCAL calling conventions to control how parameters are passed on the stack. Every window procedure has four parameters passed to it, each being one of the components that make up a message. Thus, the window procedure gets the message broken into its component parts.

When the window procedure gets control, it must first initialize any instance data it needs. Instance data, you will recall, is the data that does not need to be saved across calls to the function. In this sample program, you need a presentation space handle, a rectangle structure, and a point structure. These need not be saved across calls, because they may be different every time the window procedure is called.

Now the window procedure begins executing the message handling code. As explained earlier in the discussion of Presentation Manager program structure, this is a C language `SWITCH` statement. The target of the switch is the message identifier the window procedure has received. This message ID is tested against the messages of interest, and if it is an important one, the application window procedure handles it. If it is not important, the message is sent off to `WinDefWindowProc` for system default processing. In this program, you are only interested in the `WM_CHAR`, `WM_CLOSE`, and `WM_PAINT` messages.

If the message received was `WM_CLOSE`, the window procedure posts a `WM_QUIT` message to its own queue, which causes the message-processing loop in the main procedure to exit. The main procedure then executes its code to clean up all the resources it has allocated, such as the message queue. The program then terminates.

If the message received was `WM_CHAR`, it means that a key (or key combination) has been pressed. In this case, the window procedure looks in the message parameters to see what the key was. If the key was the `F3` key, the program sends itself a `WM_CLOSE`. Thus, the `F3` key causes the program to terminate.

The second `USHORT` value of message parameter 2 contains the virtual key code that was pressed. (This is extracted by the helper macro `SHORT2FROMMP`.) If it is equal to the value `VK_F3`, the window procedure has found what it was interested in and takes action on it. If not, no action is taken. (`VK_F3` and all associated definitions are in the header files.)

The `WM_PAINT` message is the most involved message handled in this program. The `WM_PAINT` message is sent to a window any time the window needs to be repainted. This can happen when you create a window, uncover a window by moving or sizing another window, size the window, or have the program writing to the window change some of the window's contents. The way the `WM_PAINT` must be satisfied is with the APIs `WinBeginPaint` and `WinEndPaint`. These two always go together and must be part of `WM_PAINT` processing.

The `WinBeginPaint` call returns a presentation space handle that is used to draw into the window. You draw everything in a window using a presentation space. In this program, the window first is filled with the color pink. The `WinFillRect` call is used to do this. Then a point is established inside the window where the text is to be drawn. Feeding this point along with a pointer to a text string to `GpiCharStringAt` draws the desired text in the window. The `WinEndPaint` call finishes the paint processing.

As you can see, handling these messages is a simple enough process that it can be executed right inside the window procedure. Many functions you execute will be this simple, although there are functions that require more time-consuming processing. For these functions you either start another thread to complete the processing or take some action to inform the user to wait until the processing is complete. The latter should be avoided as much as possible, but for such tasks as file I/O, for example, making the user wait is sometimes necessary.

Finally, if the message received was not of interest to this window procedure, `WinDefWindowProc` is called to perform the system's default action for that message. In this case, a direct call is made to that function, passing the same parameters received in the window procedure. `WinDefWindowProc` performs the default processing of the message, which is usually to set the reply to `NULL`.

That is all there is to a simple Presentation Manager program. It is quite a bit longer than traditional programs, but consider the functionality you get. This sample program

is not much more than a skeleton that is common to most Presentation Manager programs. It contains some simple functions. A program like Listing 10.2 can be the basis for all your Presentation Manager programs.

This is something that has been found to be invaluable in working with the Presentation Manager programs. If you want to add a function, simply add the code to look for other messages and code your window procedure's response to them. Once the base initialization and window creation function are there, it is a simple task to increase the functionality of the window.

Summary

There are definitely some differences between what is thought of as a "traditional" program and a Presentation Manager program. The latter is lengthier but more functional. Due to the Presentation Manager structure and the way it handles messages and performs housekeeping, a modular style of programming is almost mandatory. This code is often easier to maintain and enhance than other program structures.

Window Programming Concepts

This chapter discusses the programming concepts of windows. Understanding these concepts is the most important part of Presentation Manager programming. All windows have relationships to other windows, and they have many attributes that must be understood. Many of the attributes are mutually exclusive, and the way the system handles windows relies largely on them.

OS/2 features many types of windows. Not only are there windows to display data, but there are buttons that cause immediate action, scroll bars to scroll data in the windows, and menus to display functions available to the user.

This chapter expands on the subject of window classes and explains all the parameters of the `WinCreateWindow` and `WinCreateStdWindow` calls. Also discussed here is how each of the values specified in these calls changes the behavior of the windows they create.

Object-Oriented Programming

Something that should be addressed before you go any further is the concept of “object-oriented programming” (OOP). Many people claim that the Presentation Manager is an object-oriented system with an object-oriented interface.

The four basic principles of an object-oriented system are

- object-based
- classes
- encapsulation
- class inheritance

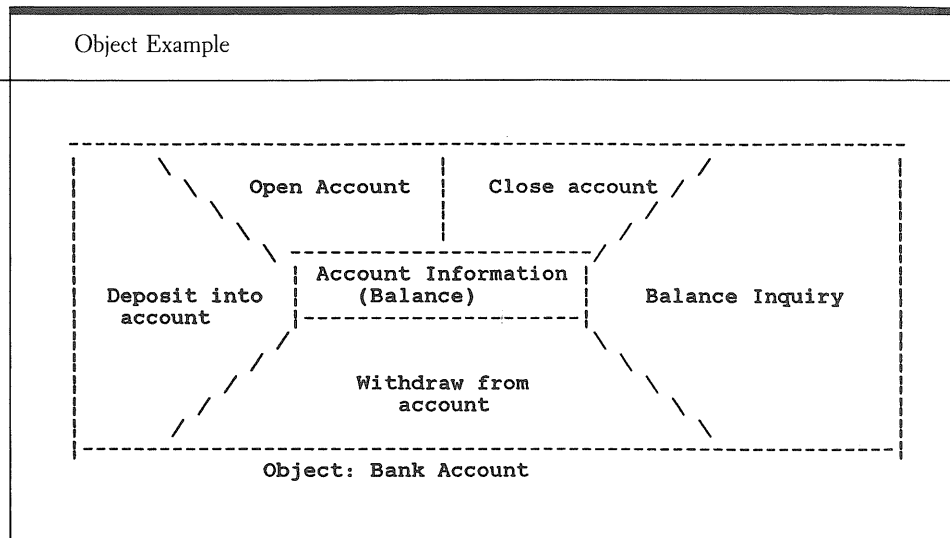
The first principle of OOP states that everything in the system is an object. An object consists of the data and a set of operations to be performed on the data. The object consists not only of the data (such as a file) but also of the operations to be performed on that data.

The second principle states that all objects belong to a class. This class defines how the object is to behave in response to a given set of stimuli (commands). Every object belonging to a class responds in exactly the same way given the same instructions.

The third principle, encapsulation, says that the only way to manipulate (or “get at”) the data is through the defined operations for that object. For example, say the data is a bank account, as shown in Figure 11.1. The operations that can be performed on the bank account are to

- open the account
- close the account
- deposit or transfer into the account
- withdraw or transfer from the account

Figure 11.1



- inquire about the balance

The data of this object (the actual account) is encapsulated inside the protective ring of operations. Only through these operations can the data be accessed. The data is protected from any direct access, and the data's integrity is maintained.

The fourth principle, class inheritance, says that when an object is created as an instance of a class, it inherits all the properties and attributes of that class. New behaviors may be added, but if an object is created belonging to some class, all of the properties of that class are exhibited in this object.

Presentation Manager and Object-Oriented Programming

The Presentation Manager system is quite similar to an OOP system in that it follows the basic guidelines. However, OOP purists would disagree that the Presentation Manager system is truly object oriented. There are some features that the Presentation Manager system allows that discount it as a "true" object-oriented system.

It is true that everything on the Presentation Manager screen is an object. The object in this case is a window combined with the window procedure that defines its behavior. Just as in the bank account example, the object consists of the data (in this case the window) and the operations that may be requested of the part of the object that defines the object's behavior (the window procedure).

Every object you see on the Presentation Manager screen is a window that has a window procedure that defines its behavior. Thus, every item on the Presentation Manager screen is an object, according to the OOP rules.

The second principle has to do with classes. This says that every item belonging to a class performs in the same way, given the same input. The Presentation Manager system contains system-defined classes that define the behavior of windows belonging to them. You may also create your own classes that define the behavior of all windows belonging to them. Presentation Manager objects are class-based, so they satisfy this requirement as well.

The third and fourth principles talk about encapsulation and class inheritance. Presentation Manager data is indeed encapsulated. The only way to get at the data (window) is through its window handle. Even then, the only thing that can be done to a window is send it messages to request it to perform operations on the window. In fact, the APIs used to manipulate windows don't do much more than coordinate the presentation of messages to the directly affected windows. The windows will then perform based on the messages.

Class inheritance is probably where Presentation Manager programs make the biggest departure from the object-oriented principles. It is true that all windows must belong to a class. It is also true that a window's behavior is defined by its window procedure,

which is the window procedure for the class it belongs to. However, there are many other variables that define a window's behavior that are not inherited from the class.

Some of these are the particular styles a window may possess, such as whether they are synchronously repainted (`WS_SYNCPAINT`). A window does not always have all the attributes of the class it belongs to. This is dynamically changeable by application programs.

The other requirement that has not been mentioned is a side-effect requirement. There must be an object-oriented language to program the system. It is where the Presentation Manager system fails the object-oriented test. As of this writing, there are few object-oriented languages and the Presentation Manager system does not use any of them. Procedural languages such as C are used to create and manipulate the objects.

Many of the aspects of the Presentation Manager system are object oriented, but without a true object-oriented language, this system cannot claim to be a true object-oriented system. The C language and the OS/2 API do a good job of providing the facilities to create an object-oriented system, but without the object-oriented language, it cannot claim to be fully "object oriented."

There have been many references to the Presentation Manager as an object-oriented system without really defining what that means. Now some of the mystery should be solved.

Window Relationships

All windows are related in one way or another. Every window in the Presentation Manager session is in a hierarchy that determines some of the aspects of its behavior. The top level in this hierarchy is called the "desktop window." The desktop window is really the whole screen, and the desktop is usually the parent or top level of the main application windows.

Each application running in the Presentation Manager session may have many windows, but each usually has only one main window. This main window is generally the addressee of messages sent to the application.

Windows are related in two ways. The first is a parent-child relationship, and the other is an owner-owned relationship. The parent-child relationship is primarily visual, whereas the owner-owned relationship is primarily structural.

Parent-Child Relationship

A parent-child relationship describes an ancestral and appearance relationship. A parent window is higher in the window hierarchy than its children and is thus closer to the desktop window. Child windows are usually created by their parent. There may be many generations of parent-child windows in an application.

An application main window is usually a child of the desktop. This main window may have child windows, which in turn may have other children, and so on. This can continue many levels down the hierarchy. A parent may have many children, but each child window must have exactly one parent.

Child windows always behave with respect to their parent. The first rule of thumb is that a child window is always clipped to its parent. This means that at no time can a child window be shown outside the physical borders of its parent. If a child window is moved so that when it is positioned its visible area lies outside the parent window, the portion outside is not visible.

The other way that children behave with respect to the parent is that whenever a parent window is moved, hidden, or destroyed, so are all of its children. This goes hand in hand with the previous rule. When a window's visibility is changed, so is the visibility of its children.

Along with the parent-child relationship is an implied sibling relationship. Windows at the same level in the hierarchy that belong to a common parent are said to be *siblings*. Siblings are only a way of describing windows; they have no bearing on each other's behavior. They may be independently destroyed, hidden, moved, and so on.

A child may change its parent with the `WinSetParent` API. This is especially useful when you want to move a window in relation to a window other than the parent originally assigned, for example.

Another aspect of the appearance of child windows is that of z-order. Child windows are always displayed on top of their parent. If a parent has only one child, the child always is the topmost window of that application. If that child has siblings, however, the z-order of the windows within the application may be determined in one of two ways.

The first way is determined by the user. If there are siblings on top of their parent, the window that the user clicks on with the mouse comes to the top. This window will also be given the focus as the user has indicated he or she wishes to work with that window. In this case, the former topmost window within the application moves back in the z-order.

The other way the z-order is changed between siblings is under program control. The most commonly used function to change the position of windows is the `WinSetWindowPos` API. This function may change the size, location, visibility, and z-order placement of any window on the screen. Depending on actions the user may take within the application, it may become necessary to bring a window on top of its siblings without having the user click on it. `WinSetWindowPos` can accomplish this:

```
WinSetWindowPos((HWND)hwndHelloFrame, /* Perform the action on this window */
                (HWND)HWND_TOP,         /* Place it on top in the z-order */
                10,                       /* Initial x coordinate is 10 */
                10,                       /* Initial y coordinate is 10 */
                300,                      /* X size is 300 */
                0);
```

```

400,                /* Y size is 400                */
SWP_MOVE|SWP_ZORDER|SWP_SIZE); /* Change the z-order and the size */

```

This function call sets the size of `hwndHelloFrame` to 300×400 placed at coordinates (10,10) and places it on top of all other windows.

`WinSetWindowPos` is a very powerful API. All the SWP flags shown here may be ORed together in the call. In addition, the function is smart about the order in which the operations are carried out. For example, say that this call had `SWP_MINIMIZE|SWP_SHOW`. `WinSetWindowPos` first minimizes the window and then shows it, thus avoiding “jumpy” screens.

WinSetWindowPos Flag	Effect
<code>SWP_ACTIVATE</code>	If the window handle specified is a frame window, make it the active window. If not, this flag has no effect on the window. The window is also made the topmost window unless specified otherwise by <code>SWP_ZORDER</code> and the ID of the window it should be placed behind.
<code>SWP_DEACTIVATE</code>	If the window specified is a frame window, deactivate it. If not, this flag has no effect. In addition, the window is made the bottom-most window in the z-order unless overridden by <code>SWP_ZORDER</code> and the behind indicator.
<code>SWP_HIDE</code>	The window specified becomes hidden.
<code>SWP_MAXIMIZE</code>	This causes the window to be maximized. If it is already maximized, the flag has no effect. This flag can't be ORed with <code>SWP_MINIMIZE</code> or <code>SWP_RESTORE</code> .
<code>SWP_MINIMIZE</code>	This flag causes the window to be minimized. If it is already minimized, this has no effect. This flag can't be ORed with <code>SWP_MAXIMIZE</code> or <code>SWP_RESTORE</code> .
<code>SWP_MOVE</code>	Move the window in accordance with the x,y parameters.
<code>SWP_NOADJUST</code>	This causes the system to refrain from sending a <code>WM_ADJUSTWINDOWPOS</code> message to the window before it is sized or moved.

SWP_NOREDRA	This causes changes in the window not to be redrawn.
SWP_RESTORE	This causes the window to be restored to its previous size and position, the values for which are stored in the window words. They can be changed by WinSetWindowUShort. The flag has no effect on the window if it is in its "normal" state. This flag cannot be ORed with SWP_MINIMIZE or SWP_MAXIMIZE.
SWP_SHOW	This causes the window to be shown.
SWP_SIZE	The window's size is changed in accordance with the cx,cy parameters.
SWP_ZORDER	The z-order placement of the window is changed with respect to the second parameter (the "behind" parameter).

Owner-Owned Relationship

The other type of window relationship is that of an owner and owned window. This type of relationship is an implied control relationship, and, in many cases, an owned window is said to be a control of its owner. As a matter of fact, most windows that do not function as controls have no owner (`owner = NULL`).

In the process of sending Presentation Manager messages, a control sends notifications to its owner. An example of this is a pushbutton control. A pushbutton is a control of its owner (because it acts solely to notify the owner of actions) and when these buttons are clicked, notifications are sent to the owner window to inform it of what the control is doing. The owner may wish to take action based on the notification, or it may wish to inhibit some actions that the control is about to take. This is the idea behind the implied control with owners and the windows they own.

Messages such as `WM_DRAWITEM` or `WM_CONTROL` are examples of notification messages. They inform the owner of the control that an event has occurred. The owner is expected either to take some action on these types of messages or to allow the default window procedure to handle them. In any case, they indicate to the owner that something is being done to a control it owns and gives the owner the opportunity to take action.

A good example of an owner relationship is a standard window with an Action Bar menu. The Action Bar menu is owned by the frame window, and when the menu is about to become active, a `WM_INITMENU` notification message is sent to the owner

(the frame). This affords the owner the chance to do some initialization processing before the menu actually becomes active. In this particular case, the menu control generates and sends this message to its owner.

There is a rule you should follow with owner windows: Any window owned by another *must* run on the same thread of execution as the owner. This is a requirement placed on applications by OS/2. Indeed, controls should really be closely tied to their owner, and this enforces that idea.

The other implication of a window being owned by another is that whenever an owner is hidden, minimized, or destroyed, so are all the windows it owns (and so on for windows they own).

Window Classes

Every window in the Presentation Manager session has its own unique characteristics. As you have seen in the outline of the simple Presentation Manager program, every window has a window class. This associates the window with a window procedure that defines its behavior. Every window belonging to a specific class is controlled by the window procedure for that class.

Public Classes

The system defines several window classes for application use. These classes control the Presentation Manager default window types. These are listed in Chapter 10. Any program may create windows belonging to these window classes. They are known as *public classes*.

Public window classes may be used by any Presentation Manager program. They are classes registered by the system at system initialization time. The window procedures for these classes reside in DLLs.

When you wish to create a window using one of these public classes, simply use the appropriate WC_ class name for the class you want in your window creation call. The public classes are registered by the system so the window procedure for each class is already associated with the class name. Each of the classes with the exception of WC_FRAME is what is called a *control class*.

Controls are used in a variety of situations—usually to provide the user with control over the behavior of an aspect of the application. Any type of window may have a control and the control notifies its owner when certain events take place. We will speak more about controls and notification messages when dialogs are discussed, later in this section. Suffice it to say that there are many public classes that define controls.

Each of the public classes' window procedures define behavior that conforms to SAA. Their appearance and behavior follow the SAA guidelines for presentation and

user interaction. Most of the system default processing that has been mentioned is performed by the procedures for these public classes. Each of the public classes has a window procedure designed specifically to control that class of window.

The `WC_FRAME` class, for example, is a complex class of window, one designed to own controls and other types of windows. The thing that makes it different from the control classes is that its window procedure is specifically designed to respond to the many types of notification messages that controls send to their owner. The frame is usually the owner of many controls and must be able to respond to these notifications to ensure they all act as one unit.

Specifically coded in the window procedure for the `WC_FRAME` class are response actions for the different types of notifications, in addition to the code to handle the presentation and behavior of the frame itself. For example, when a `WM_CONTROL` notification message is received by a `WC_FRAME` window, the frame's window procedure responds by sending the message to the client window. That is how the windows belonging to the `WC_FRAME` class are programmed to respond to `WM_CONTROL`.

A window procedure that you define to own a control will respond to whatever you tell it and ignore others. This is useful for defining your own classes that you may wish to add as a system extension using OS/2's ability to define new public classes.

These new public classes are treated the same as any other public class, accessible to any program in the system. The window procedures for the new classes are built into DLLs, as are the system-defined window procedures, and loaded by the system at initialization time.

Private Classes

Public classes are excellent for defining behavior for many common types of windows but a facility is needed to define classes of windows that you can define behavior for. Some private window classes can be defined within application programs. These classes are available only to the process that creates it and its behavior may be tailored as needed.

Most of the Presentation Manager programs you write will have private classes. Most of the data displayed in windows needs to be drawn in a specific way that is quite often different from the way the generic classes draw their data. In this case, the `WinRegisterClass` API must be used to create a class specific to the application.

`WinRegisterClass` is also used at system initialization time by the Presentation Manager shell to register the public classes that reside in the DLLs. In this instance, you will be using it to register a private window class that is accessible only to the current process. When that process terminates, all of its private classes are gone, too.

Registering the class does several things. It makes the Presentation Manager shell aware of the class so windows may be created with that class. It also associates the class name with a window procedure. In addition to the window procedure, there are several

style bits that control the behavior of the windows in a particular class. These styles are described in the following:

Window Style	Corresponding Behavior
WS_CLIPCHILDREN	This attribute specifies that child windows are to be excluded when you draw this window. The parent will not draw over where the children lie so the system cannot draw something that is just overdrawn in a second.
WS_CLIPSIBLINGS	When this style is set, sibling windows are clipped to each other's boundaries so they cannot draw something that will be overdrawn in a second.
WS_DISABLED	When a window has this attribute set, it is disabled. Windows are normally enabled. Note: A disabled window may still receive and possess the focus.
WS_GROUP	This style tells which dialog items make up a group. Set this style on the first window of a group in a dialog.
WS_PARENTCLIP	This controls clipping when drawing is taking place in this window. In general, a WS_PARENTCLIP window does not draw outside its window rectangle.
WS_SAVEBITS	This says that the screen image of the area underneath a window possessing this style should be saved when the window is made visible.
WS_SYNCPAINT	This causes the window possessing this attribute to be synchronously repainted.
WS_TABSTOP	This identifies a dialog item in which the user may use the tab key.
Class Style	Corresponding Behavior
CS_CLIPCHILDREN	All windows belonging to this class have the window style WS_CLIPCHILDREN, whether or not it is turned on in the create call.

CS_CLIPSIBLINGS	This style causes siblings to be excluded when you draw in the window. Usually, siblings are included.
CS_HITTEST	This style causes a WM_HITTEST message to be sent to the windows belonging to this class before any mouse message is sent.
CS_FRAME	This style expects all windows belonging to the class to behave as frame windows.
CS_MOVENOTIFY	This is used when a child window of this class wishes to be notified with a WM_MOVE message.
CS_PARENTCLIP	All windows belonging to this class have the window style WS_PARENTCLIP whether or not it is turned on in the create call.
CS_PUBLIC	This causes a public window class to be created. This is a special class type and may not be executed by an application program. This may only be executed from within a DLL at system initialization time.
CS_SAVEBITS	All windows belonging to this class have the window style WS_SAVEBITS, whether or not it is turned on in the create call.
CS_SIZERDRAW	Determines whether a window in this class should be redrawn when sized. When the size of the data to be drawn in the window changes with the size of the window, use this style. If the data is merely clipped to the window rectangle when the window is sized smaller, for example, this is not necessary.
CS_SYNCPAINT	This causes all windows belonging to this class to be synchronously repainted.

Whenever a class has a particular class style, all windows belonging to that class have the attributes associated with it. For example, a window belonging to a class with the CS_SYNCPAINT style all have the WS_SYNCPAINT style even if they were created without this attribute explicitly.

Once a class is registered, windows belonging to it can be created. There may be a number of windows that belong to a single class. They all share the same window procedure. Each window that belongs to a particular class is called an *instance* of that class. Each instance may have its own characteristics, but underneath it all, they all share the same window procedure.

An example of different instances of a class might be a window in which there are two pushbuttons, one representing "Enter" and the other representing "Cancel." Each of these buttons is an instance of the `WC_BUTTON` class, and as a result, each is controlled by the one window procedure associated with that class. However, there are several differences between the two instances.

The first difference is the most visible: the window text. Inside one button, the string "ENTER" is displayed; "CANCEL" is displayed in the other. Also, when this particular window was designed, it was decided to make the default button the "Cancel" one. This window has the `BS_DEFAULT` style and has its border drawn in the highlighted state.

Each of the two windows has individual characteristics, but when the user takes action in either one, the same window procedure receives the message and reacts accordingly. The parameters of the message differ in content (for example, the identifier of the window selected) but the message is the same.

Using `WinRegisterClass`, classes may be created for the client area of a standard window, classes for new controls, or even classes for windows that just process messages and don't handle any data display at all. The public classes are there for application use and need not be registered by applications wishing to use them. There is also a facility called *subclassing*, which may be used to modify the behavior of a window belonging to a class. This topic is discussed in depth in Section 6.

Basic Versus Standard Windows

Now that the classes have been defined, the next logical step is to create windows belonging to these classes.

The definition of a window is a rectangular subdivision of the screen upon which data is displayed and input from. Each Presentation Manager window has different attributes and, accordingly, different behavior. What is thought of as a window is really a "standard" window. A basic window is a single rectangular subdivision of the screen. What is termed a standard window is a complex network of windows that appears and acts as a cohesive unit.

The reason there are standard windows and basic windows is that each serves different purposes. The standard window is geared for user interaction, whereas a basic window may be finely tuned to display data or process messages in many ways without requiring user interaction.

Listing

Listing

The Basic Window

The basic window is created with a call to `WinCreateWindow`. This is shown in Listing 11.1. This basic window may belong to any of the classes publicly defined by the system or any private class that you may register.

Listing 11.1

```
#define      ID_WIN    1                /* Window identifier */
CHAR        szClassName[]="MyClass Name"; /* Window class name */
HWND        hwndMyWindow;              /* Window handle */

hwndMyWindow = WinCreateWindow(HWND_DESKTOP, /* Desktop is the parent */
                                (PSZ)szClassName, /* Specifies the class name */
                                (PSZ)NULL,         /* No window text */
                                OL,                 /* No other window styles */
                                10,10,50,50,       /* Size 50 x 50 at (10,10) */
                                NULL,              /* No owner window */
                                HWND_TOP,          /* Place on top of all others */
                                ID_WIN,           /* Window Identifier */
                                NULL,             /* No control data flags */
                                NULL);            /* Reserved. Must be NULL */
```

This window has the desktop as the parent, the class name specified in `szClassName`, no special style bits set, an initial size of 50×50 located at (10,10), and no owner. It is placed on top of all siblings and has the `ID_WIN` identifier. The handle is passed back in `hwndMyWindow`.

The `WinCreateWindow` API functions exactly as its name implies. It creates a single rectangle on the screen. It may not be visible, depending on whether the `WS_VISIBLE` style was specified.

There is a common misconception about the `WinCreateWindow` API, however. There is no rule that says that windows belonging only to private classes may be created by `WinCreateWindow`. It works just as well for public classes, too. Simply specify the system class name for the class name parameter and any flags and styles you may need, and off you go. The example in Listing 11.2 shows how an entry field window can be created using `WinCreateWindow`.

Listing 11.2

[illegible]

```

ID_EFIELD,      /* Window Identifier */
NULL,           /* No control data flags */
NULL);          /* Reserved. Must be NULL */

```

When this method is used rather than the resource file method for creating controls such as entry fields, you gain a great deal of freedom in determining who is made the owner and when the window gets created. You also have more control over the details, such as window styles. A major advantage of dynamic window creation over defining the control in the .RC file is that windows may be created based on information received from the user during the course of the application.

In the resource definition file, templates are specified with your instructions on what windows to create at specified locations. This can lead to problems if you want to place a button in a client window, for example. Using this function, windows may be created anywhere you want them, at any time you wish, with few or no details known before program execution begins.

The Standard Window

A standard window is really not a window at all. It is a “package” of windows that together provide a comprehensive set of functions for user interaction. Each item in the standard window is a window itself, each belonging to a class and having distinct behavior. In this window package, interaction with any one window may cause reactions from another part of the package. These windows work together to act as one.

The standard window is designed to conform to the SAA Common User Access (CUA) guidelines for user interaction. All the controls in the standard window are CUA conforming and they behave according to these guidelines. Each of the controls belongs to one of the public window classes, each of which is CUA compliant.

The primary part of the standard window is the frame. This window is an instance of the WC_FRAME class. The frame is the parent and owner of all of the controls in the standard window. The only part of the standard window that is not necessarily in a public class and is not owned by the frame is the client. The client window is a child of the frame but has no owner (owner=NULL). This is because the client is not a control of any window and has no need to send notification messages to anyone.

The example in Listing 11.3 shows the syntax of the WinCreateStdWindow call. Many more items are available in this call than in the WinCreateWindow call. There are flags to designate which controls are to be included in the frame such as a system icon and Title Bar. There is also a parameter to indicate what class the client belongs to and one to allow the system to return the client window handle.

Listing 11.3

```

#define ID_RESOURCE 1          /* Resource identifier */
ULONG myctldata = FCF_STANDARD & ~FCF_ACCELTABLE; /* Control data for window */
HWND hwndFrame;               /* Window handle */
CHAR szClassName[] = "Window Classname";          /* Window class name */

```

```

CHAR  szTitleMessage[] = "Window Test ";          /* Title Bar message */
HWND  hwndWin1;                                     /* Window handle */

hwndFrame = WinCreateStdWindow(HWND_DESKTOP,       /* Desktop will be parent */
                               WS_VISIBLE | FS_ICON, /* Visible and have an icon */
                               &myctldata,         /* Use ctrl data from above */
                               (PSZ)szClassName,    /* Class name */
                               (PSZ)szTitleMessage, /* Title Bar text */
                               0L,                  /* No special client styles */
                               (HMODULE)NULL,        /* No resources in DLLs */
                               ID_RESOURCE,         /* Window identifier */
                               (HWND FAR *)&hwndWin1); /* Client window handle */

```

WinCreateStdWindow performs its function by simply executing WinCreateWindow twice. The first invocation creates the frame and all of its controls.

The way WinCreateStdWindow is told which controls to give the frame is with a set of defined constants. These constants are the FCF_* values. These are called the *frame control flags*. Each flag causes a specific control to become part of the frame window. The flags are listed here. Inside the frame window procedure, the FCF values are examined and the controls are created as a result. Each of these values may be ORed together to create a single value that is used as the third parameter to WinCreateStdWindow (ctldata).

Some frame control flags control certain aspects of the frame, such as whether the program should be added to the task manager or whether it should be created with a size specified by the shell.

Frame Control Flag	Effect
FCF_ACCELTABLE	This flag causes an accelerator table to be loaded from the resource file specified in the WinCreateStdWindow call.
FCF_BORDER	This flag creates a thin, unsizable border.
FCF_DLGBORDER	This uses a dialog style border.
FCF_HORZSCROLL	This flag creates a horizontal scroll bar.
FCF_ICON	This flag associates an icon with the window to be used when the window is minimized. If this flag is used, the RESOURCE parameter of WinCreateStdWindow must be the ID of this icon as specified in the resource file.
FCF_MENU	This creates Action Bar control.

FCF_MINBUTTON	This flag creates Minimize button.
FCF_MAXBUTTON	This creates the Maximize button.
FCF_MINMAX	This flag creates minimize and maximize icons.
FCF_NOBYTEALIGN	When this flag is not set, the operations such as sizing are performance optimized.
FCF_NOMOVE- WITHOWNER	This causes the window not to move when its owner moves.
FCF_SHELLPOSITION	This flag causes the window to be created with a size and position given by the shell rather than the program.
FCF_SIZEBORDER	This creates a wide sizing border.
FCF_STANDARD	This is a composite value consisting of the ORed values FCF_TITLEBAR ; FCF_SYSMENU ; FCF_MENU ; FCF_SIZEBORDER ; FCF_MINMAX ; FCF_ICON ; FCF_ACCELTABLE ; FCF_SHELL- POSITION ; FCF_TASKLIST These values are ORed together. If the control data parameter is NULL, this value is assumed.
FCF_SYSMENU	This flag creates the System menu icon.
FCF_SYSMODAL	This causes the frame to be system modal. You will see in the discussion of modality that this means no other windows may have interactions while this window is active.
FCF_TASKLIST	When this flag is set, it causes the program title to be added to the frame window text. This string is also added to the task manager and removed when the window is destroyed.
FCF_TITLEBAR	This creates Title Bar control.
FCF_VERTSCROLL	This flag creates a vertical scroll bar.

WinCreateStdWindow, among other actions, calls WinCreateWindow twice to ac-

comply with the window creation. Behind the scenes, the first call to `WinCreateWindow` looks at the control flags specified in the `WinCreateStdWindow` call and creates each control with the frame window as the owner. The controls are created using the public window classes such as `WC_TITLEBAR`. It is important to note here that any resources that will belong to the frame, such as an Action Bar menu and minimize icon, must have the same resource ID, which must match the ID specified in the next to last parameter of the `WinCreateStdWindow` call. This ID is used to refer to the window as a whole. If they do not match, the frame and its resources do not operate properly. The parent of the frame is specified in the function call, and the owner of a frame window is initially set to `NULL`. This may be subsequently changed with `WinSetOwner`.

The second part of the `WinCreateStdWindow` call makes another call to `WinCreateWindow`, this time to create the client. Using the class and style information from the supplied parameters, the client is created with the frame as its parent and positioned with respect to the frame and its controls. The behavior of the client is controlled solely by its class and styles you provide. The handle to this new window is placed in the last parameter of the `WinCreateStdWindow` call.

The `WinCreateStdWindow` call returns two handles, one for the client and one for the frame. The client behaves independently of the frame. The frame and its controls customize the window's appearance according to the user's directions with respect to size, scrolling, and placement on the desktop. The client window is where the data is displayed in a standard window.

Once the standard window is created, it may be accessed for communication via the frame's handle. All messages affecting the standard window as a unit will go to the frame. This includes messages such as `WM_SIZE` and `WM_MOVE`.

Any message that comes from a frame control such as a scroll bar or menu will go to the frame. Some of these messages cause the frame to send messages to the client to notify it of important events that it should be informed of, such as menu action.

All communications with the window with regard to data presentation are directed to the client, because this is where the data is displayed. The frame window handles how the client is shown with respect to size and placement. The client receives a set of messages that the frame does not. The messages include ones to command it to paint information in this client window. Many of these messages may come from the frame. This process is discussed shortly in the topic on communicating with windows.

The `WM_CREATE` Message

During the course of window creation, the Presentation Manager shell sends many messages to different windows. One of the messages that is sent is `WM_CREATE`. This is a notification message that is sent to a window during creation. The message is sent to allow the window procedure for the window being created to perform some processing prior to the window becoming visible.

This message enables the window procedure to get control before the window creation is complete. Programmers often use the message to allow the window procedure to perform some initialization for the window being created. Using this message is as easy as responding to any other message your window procedure may receive.

```
case WM_CREATE:                                /* If the message is WM_CREATE */
    Send a message to another window          /* Do the work desired          */
    Initialize some data structures
    return(FALSE);                             /* Return FALSE to continue    */
break;                                         /* Break the case              */
```

The most important thing to remember in processing the `WM_CREATE` message is the return value. When the sender of this message receives the value `TRUE` as the return code, that value indicates that the sender should not continue with the remainder of the window creation. A `FALSE` reply tells the sender to continue with the window creation as normal. This indicator is to allow the processor of the `WM_CREATE` to finish up the window creation if desired. If it is told to finish up (the value returned is `FALSE`), the tasks done by the window procedure are still valid. The return value simply indicates that the remainder of the window creation was not performed and should be performed by the default procedure.

Window Presentation and Behavior

Now that the windows are created, how is their presentation managed? The behavior of windows is defined by the window procedure, classes, and styles, but what do these things really mean? How do you go about painting the items you wish to display in windows?

Styles

Window styles control certain aspects of windows' behavior. Styles control how windows react to certain stimuli or the states or attributes the windows currently have. The styles affect how the window is acted upon by other pieces of the system. They do not affect what you may do to a window. For example, the `WS_SYNCPAINT` style only determines when you are sent `WM_PAINT` messages. It does not affect how you may process them. Most important to remember is that styles are defined for each individual window, and certain ones, such as the current state styles (`WS_MINIMIZED`, `WS_VISIBLE`), may be changed during program execution.

Other styles that a window possesses may be as a result of the class that it belongs to or of the style bits that were set for the window at the time it was created. In the discussion on window classes, you saw that a class contains, among other elements such as its window procedure, a set of class styles that are specified with the `CS_*` values from the include files. These class styles are merely the system's way of having all windows belonging to a class inherit the styles of that class.

The different styles may be specified on a per-window basis, but to make window creation easier, class styles are provided. The class styles are a way of giving the various styles to each window belonging to a class.

A term used quite often in this discussion of styles is *window rectangle*. This is the way the size and position of a window is described in relation to the desktop.

The `WS_SYNCPAINT` window style (and correspondingly, `CS_SYNCPAINT`) controls how a window is updated. If an area of a window is made visible, it is the responsibility of that window's window procedure to paint the newly visible areas. This area is often called the *visible region* (or *viz-region*).

A window without the `WS_SYNCPAINT` style will not be painted as soon as it is made visible. Each window has an *update region*. The update region is the set of areas that the system has determined require painting. As new pieces of the window are made visible, they are added to the update region. This is also called *invalidating* areas of the window. The painting of the window is called *validation*.

When there are no messages in an application's queue and the `WinGetMsg` or `WinPeekMsg` function is called, a `WM_PAINT` message is generated for that window if it has a nonempty update region. Because there is only one update region, only one `WM_PAINT` message need be generated. This message is sent to the window procedure and causes the window procedure to paint the window.

When a window has the `WS_SYNCPAINT` style bit set, either by belonging to a `CS_SYNCPAINT` class or by having the bit set for it at creation time, the window is updated as soon as any area is invalidated. The action that uncovered the area of the window is not considered complete and its window procedure does not return until all syncpaint windows affected have been validated.

In general, application windows that may be drawn quickly should have the `WS_SYNCPAINT` style, whereas those that require more extensive processing and take longer to redraw should not. You should defer painting in these windows until the system is not as active. The `WS_SYNCPAINT` style provides this function.

Main windows of applications should usually be made syncpaint, whereas more detailed and complicated children may not be, as the situation dictates.

Windows may also be validated and invalidated using the API. You use the `WinBeginPaint/WinEndPaint` pair to validate and repaint windows as a response to `WM_PAINT` messages. `WinInvalidateRect`, `WinInvalidateRegion`, `WinValidateRect`, and `WinValidateRegion` may be used to invalidate and validate specific parts of windows directly under program control.

The `WS_PARENTCLIP` style is used by windows that do not overlap and have a common parent. This style causes the output into these windows to be clipped to the parent's viz-region. If this style is not set, the window is simply clipped to its own window rectangle.

The only time there is really a need to specify the `WS_CLIPCHILDREN` style is when it is desirable to prevent a parent from drawing anything on its children. If both

the child and parent are invalidated, the parent is drawn first, followed by the child. It is senseless to have the parent draw on top of the child only to have the child refresh itself over what the parent just did. This is a case where the `WS_CLIPCHILDREN` style comes in handy. The role of the `WS_CLIPSIBLINGS` style is analogous to that of the `WS_CLIPCHILDREN` style. You use it when you do not want siblings drawing over each other when they become invalidated.

In all cases here, it should be stressed that these styles do not determine window integrity. Windows are displayed correctly when all drawing is completed. The primary reason to have different types of clipping is to make drawing as fast as possible; in this system, parts drawn are not overlaid by a sibling or child. Drawing is done from the back to the front in the z-order, and by specifying these styles where necessary, you enhance drawing performance.

Painting

Painting a window is a direct response to the `WM_PAINT` message. When the `WM_PAINT` message is received by an application's window procedure, it must take some action to paint (or validate) the window. There are several reasons an application may receive a `WM_PAINT`.

- All or part of this window has been uncovered.
- The window has been sized.
- Data displayed in the window must be changed.

The `WM_PAINT` messages sent as a result of the first two actions are sent by the Presentation Manager mechanism to inform the window that it must fix itself by repainting. The painting as a result of the third action, the data display, is caused by the application itself.

When the Presentation Manager mechanism detects that a window (or a portion of a window) has been invalidated either by sizing or movement, it sets the particular area of that window invalid. Depending on the styles set for that window (`WS_SYNCPAINT` especially), a `WM_PAINT` is sent to the window. The data parameters of the `WM_PAINT` contain information about the update region of the window rectangle.

The other way an application may receive a `WM_PAINT` is as a direct result of application processing. When the application determines that it would like to redraw all or part of a window, it may use the `WinInvalidateRect` or `WinInvalidateRegion` APIs to cause a `WM_PAINT` message to be sent. When these APIs are used, the Presentation Manager mechanism takes the information provided in the API call and generates one or more `WM_PAINT` messages and sends it (them) to the proper window(s). An application cannot send a `WM_PAINT` directly.

The window procedure receiving the `WM_PAINT` message will process it by repainting the affected areas as determined by the data in the message.

It is interesting that the constant movement of the mouse across the screen and in and out of windows does not cause repainting in windows. This is handled in a special way inside the presentation drivers of the Presentation Manager interface.

Controlling User Input

During the course of running the Presentation Manager system, the user can be interacting with many applications. Obviously, the user can only be doing one thing at a time, although many applications may be running at once. There must be some way to direct the keyboard and mouse input messages to the right place. You first learned of this mechanism in the earlier discussion about the Presentation Manager architecture. What does this mean from the perspective of the application program?

Focus

The way the message router knows where to send keyboard `WM_CHAR` messages is by determining which window has the input focus. The *focus* is how the system knows which window the user is currently working with. This window is said to possess the focus and is called the *focus window*.

Usually the users click the mouse on the window they wish to work with. When a window is clicked with the mouse, that window is made active and is given the focus. Behind the scenes, there is a little more going on than you might expect.

The single button click is a system accelerator that causes a series of messages to be sent to the window currently containing the focus and the window that was clicked on that will be receiving the focus. The focus may also be changed by the system functions that change the active window. These include `ALT-ESC`, `ALT-TAB`, and the Task List. The other way these messages that cause the focus to change are sent is as a result of the `WinSetFocus` or `WinFocusChange` parameters.

The full set of messages that are sent as a result of a change in focus are

<code>WM_SETFOCUS</code>	Sent to the window losing the focus
<code>WM_SETSELECTION</code>	Sent to the windows losing their selection
<code>WM_ACTIVATE</code>	Sent to the window being deactivated (<code>MPARAM1=FALSE</code>)
<code>WM_ACTIVATE</code>	Sent to the window being activated (<code>MPARAM1=TRUE</code>)
<code>WM_SETSELECTION</code>	Sent to the windows being selected
<code>WM_SETFOCUS</code>	Sent to the window gaining the focus

Windows may look for these messages inside their window procedure to take some action when the focus is being given to them or taken away. When the focus is changed as a result of a mouse click, all these messages are sent. When the focus is changed as a result of a program calling the `WinSetFocus` or `WinFocusChange` APIs, this is not always the case.

The `WinFocusChange` API is a function that allows detailed control over the change of focus of a window. For example,

```
WinFocusChange(HWND_DESKTOP,          /* Desktop handle      */
               hwndNewFocusWindow,    /* New focus window    */
               FC_NOBRINGTOTOP|FC_NOSETSELECTION); /* Don't change z-order */
                                                /* or selected window */
```

The first two parameters of the `WinFocusChange` API are what you might expect. They are the desktop window handle and the handle of the window that will be gaining the focus. The last parameter is a combination of flags you can use to modify the messages sent as a result of the focus change. The list here shows each of the flags that may be used to change which messages get sent:

Flag	Effect
<code>FC_NOSETFOCUS</code>	Do not send the <code>WM_SETFOCUS</code> message to the window receiving the focus.
<code>FC_NOLOSEFOCUS</code>	Do not send the <code>WM_SETFOCUS</code> to the window losing the focus.
<code>FC_NOSETACTIVE</code>	Do not send a <code>WM_ACTIVATE</code> to the window being activated.
<code>FC_NOLOSEACTIVE</code>	Do not send the <code>WM_ACTIVATE</code> to the window being deactivated.
<code>FC_NOSETSELECTION</code>	Do not send the <code>WM_SETSELECTION</code> message to the window being selected.
<code>FC_NOLOSESELECTION</code>	Do not send a <code>WM_SETSELECTION</code> message to the window being deselected.
<code>FC_NOBRINGTOTOP</code>	Do not bring any window to the top as a result of the function call. Ordinarily, the new focus window is brought to the top.
<code>FC_NOBRINGTOTOP-FIRSTWINDOW</code>	Do not bring the first frame window to the top.

FC_SETACTIVEFOCUS Set the focus to the child window of that frame that previously had the focus.

Any of the flags just shown may be ORed together to form a composite value. If the value of this parameter in the WinFocusChange API is 0, every focus change message is sent. This is precisely the function of the WinSetFocus call. This function is nothing more than a WinFocusChange message with the focus change flags set to 0. This function call is

```
WinSetFocus(HWND_DESKTOP,hwndMyWindow); /* Change the focus to hwndMyWindow */
```

When the focus window is changed, all keyboard input goes to this new focus window until such time as the focus is changed to another window.

Capture

Under normal conditions, mouse messages go to the window below the mouse pointer on the screen. This mapping and translation is performed by the message router. On occasion an application may wish to have all mouse input routed to it for some period of time. The facility that the Presentation Manager shell provides to accomplish this is called mouse "capture."

You can call the WinSetCapture function to capture the mouse. When this function is executed, all future mouse messages go to the window specified in the WinSetCapture call, whether or not the mouse pointer is over that window. The capture is also released by the WinSetCapture call, but with the capture window handle set to NULL. An example of this follows.

```
result = WinSetCapture(HWND_DESKTOP,hwndCaptureWindow);
```

This code fragment sets the mouse capture to the window specified by hwndCaptureWindow. The following API call releases the capture previously set.

```
result = WinSetCapture(HWND_DESKTOP,NULL);
```

Once the capture has been set, all mouse messages are sent to hwndCaptureWindow. It is important to remember that this is a system in which a badly behaved application can cause problems for a user. If an application has the capture set to one of its windows, no other application will receive mouse messages until the capture is released. This prevents users from using the mouse to switch to other applications as well.

When the capture is released, a WM_MOUSEMOVE message is posted to the window below the mouse pointer to reactivate its mouse message processing, just as if it had been receiving mouse messages all along. The reactivation is done whether or not the mouse has moved. This ensures that the window below the mouse pointer has a chance to see the mouse message and perform any actions it needs, just as if the mouse had been moved over the window.

Setting the mouse capture is useful when you wish to prevent the user from switching away from your window. You can use it during time-critical operations where using the mouse to switch windows may be detrimental to your program. It can also be used when you wish to track the mouse movement outside your own window to allow the user to work with other objects that you control.

Communicating with Windows

Although there are APIs to manipulate windows, the only way to get information to or from a window is with messages. The APIs are really just a mechanism that sends coordinated sets of messages to various windows according to the parameters provided. Messages can be sent to windows individually under direct application control.

Each window in the system has a unique handle and a window identifier. This is how all messages sent to windows are addressed. There are also facilities available to obtain a window's handle if it is not readily available.

Once the window's handle is available, messages may be sent to set information for or query information from it. There is one fact to remember about sending messages to windows (especially ones you did not write the window procedure for). Windows only respond to messages their window procedures are programmed for. If a window does not respond to a message in the way you would like, it is not necessarily a problem of yours, it may be how the window procedure for that window is defined. This is especially true of windows of other applications.

A rule of thumb is: If a window is not yours, don't mess with it. It's perfectly acceptable to query the state of another window. You should avoid attempting to manipulate windows of other applications, however.

Identifying Windows

You know that every window has a handle, which is how messages are addressed to them. You also know that when you create a window, you are returned its handle. There are many windows that will be in your applications that you are not directly given the handles for.

When a standard window is created, for example, the frame's handle and the client's handle are returned to the program. There are many other windows that are created in this process whose handles are not returned. What is the handle for the Action Bar, for example? What about when a dialog is created from a resource file? The dialog window's handle is returned, but what about the list box and pushbutton handles? The handles are in existence, but they have not been given to the program explicitly.

There is nothing wrong with communicating with any window you wish to manipulate. You may wish to dynamically add an item to the Action Bar or change the state

of a button. The Presentation Manager architecture provides a way of obtaining the handle of any window in the system.

The `WinWindowFromID` function comes in handy when a window's handle is needed. The parameters to this call are a window and the window identifier of the desired window. For example,

```
hwndHandleWeWant = WinWindowFromID(hwndParent,  
                                     ID_WINDOW);
```

`WinWindowFromID` returns the handle of the child window of `hwndParent` that has the identifier `ID_WINDOW`. This is a restriction of this function; it may be used only when the parent of the window you want is known. This function can be used repetitively, however, so that as long as the top level window's ID is known, you can navigate down through the hierarchy until you get to the window you want.

When you want to obtain the handle of a window you did not directly create, such as a frame's Action Bar, a set of system-defined identifiers may be used. There is a set of frame identifiers (`FID_*`) that the Presentation Manager shell uses when it creates frame controls. Every frame in the system that has these controls has the same identifiers for them. As long as they do not belong to a common parent (frame) there is no need to worry about conflicts. The `FID_*` values are listed here.

Value	Control
<code>FID_CLIENT</code>	Client window
<code>FID_HORZSCROLL</code>	Horizontal scroll bar
<code>FID_MENU</code>	Action Bar menu
<code>FID_MINMAX</code>	Minimize/maximize icon
<code>FID_SYSMENU</code>	System icon menu
<code>FID_TITLEBAR</code>	Title Bar
<code>FID_VERTSCROLL</code>	Vertical scroll bar

Using these values and `WinWindowFromID`, the handle of any window belonging to the frame may be obtained. Say, for example, that you wish to add an item to the Action Bar menu. You would need to send messages to this window to get some information and then send another message to add the item to the menu. The first step is to get the handle so that you may send these messages. This is accomplished by the following code:

```
hwndActBar = WinWindowFromID(hwndFrame,  
                               FID_MENU);
```


A question comes up, “What if this window has no Action Bar?” “What if I try to obtain the handle of a window that I’m not sure exists?” If `WinWindowFromID` returns `NULL` as the window handle, that means the window was not found. Anything other than `NULL` in this value is the window handle that was returned.

Assume for this example that you want to send a message to a specific menu item. A menu item is also a window and has a handle. The value returned above may be taken, and using the identifiers specified for the menu in the resource file, may navigate through the menu levels to the desired item. With that handle now available, messages may be sent to it. Here is code for using the Action Bar handle to navigate the menu structure to such a particular item.

```

hwndSub1 = WinWindowFromID(hwndActBar,
                           MID_MYSUBMENU1);      /* Get Action Bar submenu */

hwndTargetItem = WinWindowFromID(hwndSub1,
                                MID_MYMENUITEM); /* Get handle of the menu */
                                                    /* item you want */

```

Once the handle has been obtained, you can send messages to perform the desired functions. There are two ways an application can communicate a message to a window: sending and posting.

Sending Messages

Sending a message is really a misnomer. It was given its name for conceptual purposes. When your program “sends” a message, it is really just making a synchronous call to the window’s window procedure. Nothing is placed in a queue when a message is sent. This simply allows a program to call a window procedure with a specific handle and data values to affect a given window.

The `WinSendMsg` API causes a message to be “sent” to a window procedure. This function is very similar to the `WinDispatchMsg` function discussed earlier. `WinDispatchMsg` takes a `QMSG` structure off the application’s message queue and breaks it up into its four parts. It then in effect creates a `WinSendMsg` that calls the target window procedure. This is a synchronous call, so control does not return until the window procedure finishes its processing and returns control to its caller.

The `WinSendMsg` function starts out with the four parts that make up a message as four parameters to the window procedure call. It takes these values, makes the call to the window procedure, and waits for control to return. Once again, you can see why messages or calls to the window procedure must be processed expeditiously. If they are not, control does not return to the caller and processing can start to slow down.

When you use `WinSendMsg`, the system handles the thread switches necessary to perform the call to the window procedure. The thread that the window runs on is not

always the same as the thread sending the message, so quite often a thread switch is required. OS/2 takes care of this when `WinSendMsg` is used.

`WinSendMsg` is appropriate for actions that require return values. These are functions such as `WM_QUERYWINDOWPARAMS` or `WM_FORMATFRAME`. Many other messages need only to be posted.

Posting Messages

When a message is posted, it is placed on the message queue for the window. It is an asynchronous function. The originator of the message does not wait for a function to return control; it simply posts the message and continues on its way. This message eventually is removed from the queue via `WinGetMsg` or `WinPeekMsg` and subsequently processed.

Messages are posted as a result of user input, timers (`WM_TIMER` message), or notifications from controls. These are usually messages that do not force the originator to wait until messages are processed. They do, however, need to be synchronized.

Messages of the same priority are removed from the queue in a time-ordered fashion. This chapter has outlined the priority of messages. For example, all user input messages such as `WM_CHAR` and `WM_BUTTON*` messages are removed from the message queue in a first-in first-out order. This ensures that the user input is processed in the order in which it arrived.

Messages may be posted to windows by applications using `WinPostMsg`. The parameters are the same as in `WinSendMsg`. The main difference is that `WinSendMsg` actually makes a far call to the window procedure, whereas `WinPostMsg` places the message in the window's message queue and returns immediately.

The message that is posted remains on the queue until it is removed with `WinGetMsg`. In the message loop in the main procedure, the next function is `WinDispatchMsg`, which, as you have seen, will take that posted `QMSG` structure, break it up, and call the window procedure.

Sending and posting messages is the way your programs talk to windows and manipulate window data. Going back to the discussion on object-oriented programming, the window and window procedure are the objects, and the operations are defined in the window procedure. Messages are the way to perform operations on them.

Summary

Windows are complicated objects. You have seen how each window has different attributes and relationships that cause each to behave in its own, unique way. You have seen the different ways to communicate with windows. In reality, there is only one way—messages—but conceptually, there are many means.

The APIs available are ways of sending a series of messages to perform more complicated functions than a single message may accomplish. The window handle is the most important attribute of a window.

All communications with a window require the use of its handle. Once that information is obtained, almost anything is possible.

Finally, you have seen how the different methods of communicating messages to windows work. You have also seen why messages are posted and sent and can see more clearly why messages must be processed quickly.

The following chapters explore other types of windows and how you can tailor them to your specific applications.

Messages

All this talk about messages has left many questions unresolved. How do you know how to handle these messages? How should your windows respond? How do you get the return values from messages you send?

In this chapter all these questions are answered, and examples demonstrate how windows are expected to behave. You will see how to manipulate windows and finely tune their behavior through responses to the various messages that windows may receive.

You can send many types of messages to a window. There are *notifications* an owner may expect from a window it owns. Another type is *control messages* that a window may send to another to cause a certain response. *System-generated messages* tell windows about external events that they must be aware of and, of course, *user input messages* indicate mouse or keyboard activity.

How can windows be programmed to react to the huge number of messages they may receive? Every message a window gets is important, and a message that goes unprocessed can cause windows to behave inconsistently.

The key to answering all these questions lies in the default processing the Presentation Manager window procedures provide. In the discussion of classes, you saw that every class has a window procedure that defines its behavior. Each of the system-provided classes has a system-provided window procedure. This really means that the windows belonging to these classes respond to a set of messages in a specific way. This is the definition of their default behavior.

This chapter explores ways to work with the default behavior and use it to your advantage. You will see what types of messages windows may expect and ways to respond to the messages so your code can perform advanced functions more easily.

Also, in working with these messages and knowledge of the event-driven architecture of the Presentation Manager system, you will be able to understand how and why the Presentation Manager window procedures do what they do and how to make this work for you.

Building a Window Procedure

All messages must be processed. Each one has special significance, and even though a window may not be interested in a particular message, a response is expected by the sender. Presentation Manager provides a special “generic” window procedure, `WinDefWindowProc`, that responds to any message a window may receive. This processing does nothing fancy; it simply performs some basic function and sends a return value to the sender of the message.

Through `WinDefWindowProc` and the other more class-specific window procedures you may take advantage of OS/2's default processing for windows. In application window procedures, you have seen that all a window procedure needs to do is look for the messages it is interested in and act on them while sending all others to `WinDefWindowProc`.

Every message an application receives requires a response. Some generic responses are provided through `WinDefWindowProc`, but everything that makes an application what it is will be done in the application's window procedure(s) responses to messages.

The simplest window procedure possible is

```
MRESULT FAR PASCAL HelloWndProc( hWnd, message, lParam1, lParam2 )
HWND    hWnd;
USHORT  message;
MPARAM  lParam1;
MPARAM  lParam2;
{
    return(WinDefWindowProc( hWnd, message, lParam1, lParam2 ));
}
```

When a window is created using this window procedure, all messages sent to that window are handled by `WinDefWindowProc`. A window with this window procedure will function in the most general way. Only simple frame interaction such as sizing, movement, and closure are supported. No painting will take place and no menu interaction is possible. This window will exist and nothing more. The important thing to note is that every message sent to it will be responded to.

When you build an application's window procedure, you can start off with this basic framework. Then, according to how your application's windows are designed, you start adding responses to messages.

The first step is to place the C language SWITCH statement into the window procedure and make the default processing of that switch the code shown here:

```
MRESULT FAR PASCAL HelloWndProc( hWnd, message, lParam1, lParam2 )
HWND    hWnd;
USHORT  message;
MPARAM  lParam1;
MPARAM  lParam2;

{
    switch (message)
    {
        default:
            return(WinDefWindowProc( hWnd, message, lParam1, lParam2 ));
            break;
    }
}
```

Once the SWITCH statement is in place, the cases for the messages of interest are added. For example, say you don't want button 2 to be active at all in this main window. Any click on button 2 in this window should sound a beep. A case can be added for the WM_BUTTON2DOWN message to execute a DosBeep instruction.

```
MRESULT FAR PASCAL HelloWndProc( hWnd, message, lParam1, lParam2 )
HWND    hWnd;
USHORT  message;
MPARAM  lParam1;
MPARAM  lParam2;

{
    switch (message)
    {
        case WM_BUTTON2DOWN:      /* If button 2 is pressed,      */
            DosBeep(500,700);      /* sound the beep              */
            return(TRUE);          /* and return the value to show the */
            break;                 /* message was processed        */

        default:
            return(WinDefWindowProc( hWnd, message, lParam1, lParam2 ));
            break;
    }
}
```

Now you have a window that beeps when the user clicks mouse button 2. Although this is a posted message, a value should be returned. The system may post this message to the window, but some other application may send this to the window and expect a reply. It is a good practice to have a return value for every message your window

procedure handles. Each message that you handle in your window procedures is unique in what the sender expects.

From here on, you may add responses to messages that you feel are important to the application. For each message that you find important, simply add a “case” for it in the switch. Whenever the message you are casing for gets to the window procedure, program execution begins at that case point and continues until a “break” is encountered. Between the case and the break, all of the important work gets done. Any type of instruction may be placed there. Threads may be started, I/O may be executed, or messages may be sent to yourself or others.

Here your code’s processing must be completed within the time guideline, or you risk slowing down user input. The time between when the window procedure gets called and the time it executes either the return or the break from a message must be within 1/10 second. If, because of the length of processing required as a result of a message, the window procedure cannot fulfill the time requirement, it must either start a thread to perform the processing or inform the user to wait until the operation is done.

The former is the preferable course; the latter is sometimes necessary. In the cases where it is possible to continue asynchronously, the application may start an asynchronous thread to complete the processing. This enables the window procedure to return control while concurrently allowing the application to finish its processing as a result of the message.

In case it cannot continue and it is necessary to have the user wait until the current processing has completed, the application should change the pointer to the hourglass to inform the user to wait. If this type of action is not taken, the user may be led to believe the application has crashed because it is not responding to new input.

As you have seen, the Presentation Manager message architecture enables this to happen when processing takes a long time. The way to indicate to users that, “Yes we know we are holding things up for a minute. Please stand by,” is to change the pointer to the hourglass. This is a system-defined pointer with the identifier `SPTR_WAIT`.

The code here shows how an asynchronous thread may be used to complete message processing.

```
case WM_MESSAGE_THAT_REQUIRES_LONG_PROCESSING:
    /*****
    /* Set up the stack for the thread. */
    /* If it fails, return the error */
    *****/

    if (DosAllocSeg( STACKSIZE, (PSEL)&seIT1Stack, 0 ))
    {
        return(FALSE);
    }
```

```

/*****
/* Set up the pointer to the stack */
*****/

OFFSETOF(pbThread1Stack) = STACKSIZE-2;
SELECTOROF(pbThread1Stack) = selT1Stack;

/*****
/* Kick off the thread */
*****/

retcd=DosCreateThread(Thread1Main,&thread1return,pbThread1Stack);
if (retcd != 0)          /* If the thread creation failed */
{
    DosBeep(400,200); /* beep to indicate error */
}

/*****
/* If all goes well, return the proper value for the message */
*****/

return(Proper_return_value);

break;

```

Adding sections individually for each message that is expected is how you build a window procedure. Each window defined in the application has its window procedure built in such a fashion. In addition to the procedures built solely for an application, system-defined procedures may be modified through the use of subclassing. This concept is discussed in Section 6.

Now that you have seen how to build a window procedure, take a look at how these window procedures are expected to respond to and handle messages.

Handling Messages

Message Return Values

Message return values are usually an indication to the sender of a message telling it what action was taken as a result of the message. In some cases, it is a TRUE or FALSE value (BOOL) and in others it is some kind of data structure (when a query type message is sent). Others require and expect no return value. It is important, however, that the window procedure always return some value. Whether this value is used by the sender is of no consequence. A return value should always be used, because senders that need the value do not work properly without it, whereas those that don't require it simply discard it.

When messages are posted, the message sender continues without waiting for a reply. That is the purpose of posting versus sending a message. In most cases, however, messages are sent and the reply value is very important to the sender. Messages are often sent to query the status of some aspect of a window. Although the return values from these messages contain the requested status or other relevant information, in many cases they are used to indicate failure or success.

Remember that the act of sending a message is really just a way of describing a synchronous call to the window procedure. As with all functions that return a value, the caller of the function expects a return value. This return is how the caller knows what happened as a result of the message. It is very important to return the correct values for messages, because they affect how other windows are processed.

Each message has a specific return value; depending on this value, a message may cause a reaction from other windows in the application. An example is the `WM_INITDLG` message, whose technique of message processing is shown in the following code. The `WM_INITDLG` message is sent to the owner of a dialog as a notification when the dialog is created. This message is designed to afford the owner the opportunity to perform some preprocessing on the dialog before it becomes visible.

```
case WM_INITDLG:
    Set the default button;
    Call the function that fills a listbox with items;
    return(TRUE);
break;
```

The important thing to note for this discussion is not what is being done to the dialog in this processing, but the return value. The `WM_INITDLG` message is always sent to the dialog. It can be handled by the dialog itself, or the dialog can let `WinDefWindowProc` handle it. As part of the `WM_INITDLG` default processing by `WinDefWindowProc`, the value returned is `FALSE`. This indicates to the system that no preprocessing was performed on the dialog, so the dialog creation processing continues.

In this example, the `TRUE` value was returned. This indicates to the default procedure that the owner did perform some action as a result of `WM_INITDLG` and that the procedure should not perform all of its default processing. If `FALSE` were returned, the system would think that the owner had done nothing and would set all of the dialog box settings to the system default, overriding what was done in this code.

This is only one example of how the return values from messages affect other window processing. Each message has its own interpretation of the possible return values.

WM_COMMAND

The `WM_COMMAND` message is posted by a control as a result of the control (menu, pushbutton, and so on) wishing to notify its owner of an event. The message

itself is only an indicator to the window procedure that the event has occurred. The real information is contained within the message parameters of the `WM_COMMAND`.

The values that are contained in the `WM_COMMAND` parameters are

- | | |
|----------------|--|
| MPARAM1 | This contains a single <code>USHORT</code> value indicating the command value. For example, this would be a menu item identifier if a menu item selected is the cause of this message. If a button had been selected, this <code>USHORT</code> would contain the button ID mnemonic defined for it. |
| MPARAM2 | This contains two values: <code>USHORT</code> and <code>BOOL</code> . The <code>USHORT</code> identifies the type of control that generated the <code>WM_COMMAND</code> . The possible values are <ul style="list-style-type: none">• <code>CMDSRC_PUSHBUTTON</code>, in which the source of the message is a pushbutton. <code>mparam1</code> is the ID of the button.• <code>CMDSRC_MENU</code>, in which the source of the <code>WM_COMMAND</code> is a menu control and <code>mparam1</code> is the identifier of the menu item.• <code>CMDSRC_ACCELERATOR</code>, which is posted as a result of an accelerator key. <code>mparam1</code> is the identifier of the accelerator command value.• <code>CMDSRC_OTHER</code>, for which the source is a control other than what the system provides. |

The other part of `MPARAM2` is a `BOOL` value that indicates where the action came from that caused the `WM_COMMAND`. A `TRUE` value indicates a mouse action; `FALSE` in this value indicates that keyboard action caused the `WM_COMMAND`.

This message is posted to the owner of the control to allow it to respond based on the information provided. Because this is a posted message, no reply is needed, and as a matter of convention, the field is specified as reserved and should be set to `NULL`.

The processing for this message involves extracting the information contained in the message parameters and acting on it. Quite often, this requires another `SWITCH` statement to test the identifier field to determine who sent the message. Say, for example, that there are menu identifiers `MID_ITEM1`, `MID_ITEM2`, and `MID_ITEM3` and button identifiers `BID_BUTN1`, `BID_BUTN2`, and `BID_BUTN3`. Menus 1, 2, and 3 correspond to play a tune, draw a bitmap, and exit the program, respectively. Buttons 1, 2, and 3 represent actions to paint the screen gray, beep once, and paint the screen pink, respectively. Also assume that these are the only functions this window performs

as a result of WM_COMMAND messages. The window procedure case for this might look like Listing 12.1.

Listing 12.1

```

case WM_COMMAND:                /* If the message was WM_COMMAND,      */
    switch((SHORTFROMMP))mp1     /* look at the value to see which item */
    {                             /* caused the message                  */
        case MID_ITEM1:          /* If it was menu item 1,             */
            DosBeep(329,300);
            DosBeep(392,350);
            DosBeep(329,75);      /* play a tune                        */
            DosBeep(349,300);
            DosBeep(392,325);
            break;

        case MID_ITEM2:          /* If it was menu item 2,             */
            pt.x = 10;           /* set a point,                      */
            pt.y = 150;
            hPS1 = WinGetPS( hWnd); /* obtain a Presentation Space handle, */
                                   /* load a bitmap, set it current,      */
                                   /* and draw it at the point specified */

            hbmMyBitMap = GpiLoadBitmap(hPS1,(USHORT)NULL,THEBITMAP,50L,85L);
            GpiSetBitmap(hPS1,hbmMyBitMap);
            fRetCode = WinDrawBitmap(hPS1,hbmMyBitMap,(PRECTL)NULL,
                                   (PPOINTL)&pt,(LONG)NULL,(LONG)NULL,
                                   DBM_IMAGEATTRS);

            if (!fRetCode)        /* If an error occurred, tell user    */
            {
                eErrorCode = WinGetLastError(hAB);
                sprintf(strarr,"Return Code from WinDrawBitmap: %x",eErrorCode);
                WinMessageBox(HWND_DESKTOP,HWND_DESKTOP,
                             strarr,"Error drawing Bitmap",0,MB_OK);
            }
            WinReleasePS(hPS1);
            break;

        case MID_ITEM3:          /* If menu item 3 was selected, */
            WinPostMsg(hWnd,WM_CLOSE,0L,0L); /* terminate the application */
            break;

        case BID_BUTTON1:        /* If button 1 was selected,      */
            hPS = WinGetPS( hWnd); /* obtain a presentation space handle */
            WinFillRect(hPS,      /* fill the rectangle with gray    */
                       (PRECTL)&rect,
                       (LONG)CLR_GRAY);
            WinReleasePS(hPS);    /* Release the PS                  */
            break;
    }

```

```

case BID_BUTN2:          /* If button 2 was selected,      */
    DosBeep(400,500);    /* sound a single beep on the speaker */
    break;

case BID_BUTN3:          /* If button 3 was selected,      */
    hPS = WinGetPS( hWnd); /* obtain a Presentation Space handle */
    WinFillRect(hPS,      /* Fill the rectangle with pink */
        (PRECTL)&rect,
        (LONG)CLR_PINK);
    WinReleasePS(hPS);    /* Release the PS */
    break;

}                          /* End the switch on the mp1 ID value */

break;                    /* Ignore any other type of WM_COMMANDs */

```

Inside the case for the WM_COMMAND, any mechanism you want can be used to check for the individual control identifiers of interest. This example uses another SWITCH statement. This keeps uniformity throughout the procedure and makes reading the code a bit easier. In each case inside the switch, IDs of interest are tested, and if present, the required processing is performed, much like the switch on the message IDs.

The WM_COMMAND message contains the ID of a control and other relevant information regarding that control's selection, such as what type of device it came from. It is a simple message that indicates selection only.

WM_CONTROL

In contrast to WM_COMMAND, the WM_CONTROL message is a notification message that notifies the owner of a control that something happened to the control, but the window procedure is not the primary responder to the message. In this case, the control, such as a listbox or entry field, performs actions and simply notifies their owner that an event occurred. This affords the owner window the opportunity to take some action as a result of the event. The control window, however, does the real processing.

The WM_CONTROL is more of a notification, such as "Here is what happened, do you want to do anything as a result of it?" message, whereas the WM_COMMAND represents, "Here is what happened. Do something with this because I just ignore it if you don't" message. The WM_CONTROL message is sent when a control such as a button or check box receives a mouse click and acts upon it. The owner of that control then gets the chance to act based on that action.

An example of this is a set of check boxes that represent colors the window may be changed to. You may decide to have the window change colors when the user checks a box rather than waiting for the user to click on an Enter button before the changes become visible. This may be coded as a response to a WM_CONTROL.

```

case WM_CONTROL:                                /* If the message is WM_CONTROL */
    switch((SHORT2FROMMP))mp1
    {
        case BN_CLICKED:                        /* If a button was clicked */
            if ((SHORT1FROMMP)mp1 == CBID_RED) /* If the button was RED */
            {
                hPS = WinGetPS( hWnd);          /* Get a PS handle */
                WinFillRect(hPS,                /* Fill the rectangle with red */
                    (PRECTL)&rect,
                    (LONG)CLR_RED);
                WinReleasePS(hPS);              /* Release the PS */
            }
            if ((SHORT1FROMMP)mp1 == CBID_BLUE) /* If the button was BLUE */
            {
                hPS = WinGetPS( hWnd);          /* Get a PS handle */
                WinFillRect(hPS,                /* Fill the rectangle with blue */
                    (PRECTL)&rect,
                    (LONG)CLR_RED);
                WinReleasePS(hPS);              /* Release the PS */
            }
            break;                               /* End the BN_CLICKED case */
        }
    }
    break;                                       /* End the WM_CONTROL case */

```

This code shows how a WM_CONTROL message can be detected at the owner's window procedure and cause an action to occur as a result of it. Inside the case for the WM_CONTROL message, a switch can be set up on the second USHORT value of MPARAM1 of the WM_CONTROL. This is the notification code for the message. In the first USHORT part of MPARAM1 is the control identifier to determine which control caused the WM_CONTROL.

In this instance, you want a BN_CLICKED to indicate that a button was selected. When this is the case, you want to see whether the CBID_RED or CBID_BLUE button was the one selected. If so, the program sets the color of the window to red or blue as a result. This is an example of how notifications may be used to act immediately on a control activation and not always have to wait for all controls to be selected and then process their values.

This type of operation may be used so a user can preview the results of selections before actually selecting the Enter key or button to make the changes permanent.

When controls are acted upon by the user, the control performs the work of highlighting itself, changing its appearance, and so on. That is only the visual appearance. The control sends the WM_CONTROL to the owner window to allow actions to be taken as a result. If the owner does not wish to act at that time, the user sees no reaction from the application and may continue execution. At some point, the control's selections

are read and action is then taken. WM_CONTROL is there to give owners that opportunity.

WM_PAINT

The WM_PAINT message is sent to a window whenever an area needs painting. This message can occur when the system determines that an area of a window becomes invalidated or when a window wants to change its appearance. This is a message that is always sent by the system. It gets triggered as a result of some area of the window being invalidated, either by user actions (moving or sizing) or as a result of an API call (such as WinInvalidateRect).

When this message is received, the WinBeginPaint and WinEndPaint APIs must be used. These functions are necessary to satisfy the WM_PAINT message. In between this pair of API calls, the functions necessary to repaint the window are performed.

When a part of a window becomes invalidated, an update region is created. If the CS_SYNCPAINT style is set for the window, a WM_PAINT is generated immediately and sent to the window. If that style is not set, the system holds off sending the message until the system is not as busy. In the meantime, the system combines this update region with any new invalidated areas as they occur. The values of the parameters in the WM_PAINT message call are not interpretable by the application window procedure. Issuing the WinBeginPaint call returns a handle to a presentation space (HPS) that corresponds to the update region.

This is a cached, micro presentation space. You will learn more about this in Section 5. For now, all you need to do is use that HPS in the calls to the painting and drawing functions to repaint the window with the proper data. Once this is done, the WinEndPaint function is called. This function destroys the HPS and indicates to the system that the required processing has been done to validate the update region.

case WM_PAINT:

```
hPS = WinBeginPaint( hWnd, (HPS)NULL, (PRECTL)&dummy );
WinFillRect(hPS,(PRECTL)&rect,(LONG) CLR_BLUE); /* Fill background */
pt.x = pt.y = 0L;
pt.x = pt.y = 10L; /* Set up a point */
GpiMove(hPS, (PPOINTL)&pt); /* Move to there */
pt.x = pt.y = 100L; /* Set an end point */
GpiCharStringAt( hPS, (PPOINTL)&pt, (LONG)5,
                 (PSZ)"Hello"); /* Put out message */
WinEndPaint( hPS );
```

break;

This code fills the rectangle with the color blue and then displays the string "Hello" at the point specified.

The WM_PAINT processing in a window procedure is very important, especially if you use CS_SYNCPAINT windows. If the window is not validated (painted) as a

result of the `WM_PAINT` message, you continue to receive `WM_PAINT`s until you satisfy it with `WinBeginPaint` and `WinEndPaint`. Note, however, that you do not have to actually paint the window here. You may defer painting until some future time. However, the message must be satisfied to stop the flow of `WM_PAINT`s.

A `WM_PAINT` does not require a reply. The execution of `WinBeginPaint` and `WinEndPaint` are the actions that the system needs to complete the point.

Mouse Messages

Mouse messages are received by a window whenever the mouse pointer is moving over a window or when the pointer is over a window and a button click occurs. Probably the most frequently occurring messages are the mouse messages. Most of these, however, may be ignored by your applications. They primarily consist of `WM_HITTEST`, `WM_CONTROLPOINTER`, and `WM_MOUSEMOVE` messages.

These messages are used primarily by the Presentation Manager system to determine where the mouse is at a given time and what window mouse input should be directed to. In most normal situations, the default window processing takes care of these messages. The messages that the window procedures will be interested in are the `WM_MOUSEMOVE` message and the messages for button action.

The messages that result from button clicks are

- `WM_BUTTON1DOWN`
- `WM_BUTTON1UP`
- `WM_BUTTON2DOWN`
- `WM_BUTTON2UP`
- `WM_BUTTON1DBLCLK`
- `WM_BUTTON2DBLCLK`

Each message may be acted upon individually. As the list implies, for every button action, more than one message is received. For example, clicking button 1 produces two messages: `WM_BUTTON1DOWN` and `WM_BUTTON1UP`. This is an important distinction, because it may be desirable to perform action on the upclick of a button rather than the downclick.

Another example is in the case of a double click on a button. As described in Section 2, a double click is two clicks on a single button in rapid succession. The messages that would be received as a result of a double click on button 1 are

<code>WM_BUTTON1DOWN</code>	The initial down click of the button
<code>WM_BUTTON1UP</code>	The release of the button

WM_BUTTON1DBLCLK The result of the second click occurring within the double click time span

WM_BUTTON1UP The second upclick of the button

If the application is only interested in a double click, it need not test for the down and up messages. The only concern is the double-click message. Programs don't have to concern themselves with the timing of the two clicks. The Presentation Manager input router (and translator) determines whether the second click occurred within the time set for the double click. If so, it will send the **WM_BUTTON1DBLCLK** message. If the second click was not quick enough, another set of **WM_BUTTON1DOWN** and **WM_BUTTON1UP** messages is received.

The data supplied in the parameters of the mouse messages is data pertinent to the mouse position at the time of the message. This basically consists of the mouse position and the result of the **WM_HITTEST** message. This information tells the recipient of the mouse message what it needs.

The reply value of these messages is very important. In each of these messages, a return value of **TRUE** to the sender indicates that the window procedure processed the message. **FALSE** indicates that the message was not processed. **WinDefWindowProc** always sets the return value to **FALSE**, because the default procedure takes no action on these messages. If the application processes them, however, the return value should be set to **TRUE**. This indicates to the sender that some action was taken. Quite often, the sender of a message looks for a return value such as this and takes one of several paths, depending on whether the message was processed by the addressee.

In the case of button clicks on controls such as check boxes or list box items, the button clicks need not be interpreted by the window procedure. The window procedure that defines the behavior of each control will process these. For example, it is not necessary to process button messages for menus on the Action Bar. The window procedure for the **WC_MENU** class handles that work. An application's only concern is with messages received by a window for which it has a window procedure.

When a mouse message is received by a window procedure, the window procedure is provided with the information needed to react to the mouse action. The **POINTL** structure that makes up the first **USHORT** value of **MPARAM1** gives the mouse position relative to the origin, or lower left corner of the window. The second **USHORT** value of the first message parameter is a value that tells the result of the **WM_HITTEST** message that was sent to the window to find out what type of processing (if any) is required. Using this information, the message may be processed if desired. In this case, the window procedure should return **TRUE**. If, however, when the message parameters are examined and the application determines that it does not wish to take any action as a result, it should return **FALSE** to indicate that no processing took place as a result of the message.

Mouse message processing by a window is useful for applications that use the client area of a window for data manipulation. A drawing program is a perfect example. A single click may draw a point, whereas a click and drag might be used to draw a line. Usually, a window is not concerned with mouse messages. The controls the user interacts with handle mouse messages directly.

Keyboard Messages (WM_CHAR)

Although there may be many applications with many windows on the display screen at any given time, keyboard input is not as hard to manage as it might seem. The system manages who gets the keyboard input messages through WM_CHAR. At any given moment in time, only one window on the screen may have the input focus. This focus determines where keyboard input is directed.

You can give any window on the screen the focus. To do so, click on a window with the mouse, use menus to make a particular window active, or use a system accelerator such as ALT-ESC or ALT-TAB. Once a window has the focus, all keyboard input is sent to that window until the focus window is changed.

Keyboard input goes through a set of translation routines described in Section 3. The single input queue/message router receives raw keyboard codes from the keyboard device driver. They take that information and turn it into a WM_CHAR message with a set of parameters that has been mapped into a standard set of codes.

When the WM_CHAR message gets to the window procedure, it must test the parameters for the key codes of importance. The structure of the message parameters for the WM_CHAR message is shown here.

parameter 1

USHORT - keyboard control flags

- KC_CHAR - The "ch" code field should be used.
- KC_SCANCODE - The "scancode" field should be used.
- KC_VIRTUALKEY - The "vk" field should be used.
- KC_KEYUP - The event is a key up message. If this is not set, it was a key down event.
- KC_PREVDOWN - The key was previously down.
- KC_LONEKEY - The key was pressed without any others.
- KC_SHIFT - The shift state was active when the key was pressed.
- KC_ALT - The ALT key was active (pressed) when the key was pressed.
- KC_CTRL - The CTRL key was active when the key was pressed.

UCHAR - repeat

This is the repeat count for the key that has been pressed.

e client
mple. A
a line.
e user

screen
n. The
At any
focus.

window
system
keyboard

tion 3.
keyboard
ge with

test the
imeters

```

        post myself a WM_CLOSE;          /* is not on          */
        return(TRUE);                    /* Indicate the keystroke */
    }                                     /* was processed        */
    break;

}

return(FALSE);                          /* Otherwise, indicate   */
                                          /* the keystroke was     */
                                          /* not processed         */

break; /* End of WM_CHAR message processing */

```

As you can see, all that is of interest in this application are the Enter key and F3 to cause the program to terminate. As you work through the implementation of a particular application, you can add in the code to look for different keys and code the reaction to those keys accordingly.

There are many flags associated with the WM_CHAR message. Depending on which type of keystroke is of interest, the KC_CHAR, KC_VIRTUALKEY, or KC_SCANCODE flags may be checked. These flags indicate which parts of the message parameters are valid for this particular message. For example, if the KC_CHAR flag was set, (SHORT1FROMMP)mp2 will enable you to determine the keystroke information. Each flag determines what type of keystroke information is contained in the message parameters.

When you look for keys in window procedures, you must remember that just like mouse messages, every down stroke has an up stroke. Therefore, two WM_CHAR messages are received for every keystroke (the press and release of a key or key combination). This presents a bit more to test for but also gives applications the freedom to process certain items on the down or up stroke of a key.

This can be determined by testing for the KC_KEYUP flag in the WM_CHAR message. If this flag is set, the message has been sent as a result of the up stroke of the key. Otherwise, the message came as a result of the key being pressed down.

When key combinations are important, you can check the flags to see if any of the KC_ALT, KC_SHIFT, or KC_CTRL flags are set. These indicate which (if any) of the “extra” state keys were active when the key was pressed. For example, if you wish to use the ALT-W key combination to switch between windows of an application, you would look for the W key. If this was the key pressed, look at the flags to see if the KC_ALT flag was set. If so, process your window switch and return TRUE. Otherwise, take no action and return FALSE to indicate that no action was taken as a result of the message.

Every key that is significant to your application should be processed in this way. Notice, however, that this is for windows that you are controlling. For the system-defined windows, you do not have to bother with this processing. A perfect example is an entry field. The window procedure that defines the behavior for the WC_ENTRYFIELD class performs all of this processing for you.

The keyboard messages go only to the focus window, so you don't have to worry about getting a stray message here. The Presentation Manager input mechanism handles this. All you need to do is look for the keys that mean something to your program and allow the rest to go through.

Summary

Everything in the Presentation Manager is accomplished with messages. Messages are how windows get user input, display output, and communicate with each other. You have seen how to build a window procedure from scratch and why the 1/10 second rule is so important. All messages are the same in structure, so what you do with each can easily be customized for any message.

You have also seen examples of the fundamental mouse, keyboard, and paint messages.

Handling messages is where the code to handle all your display and interaction management will go.

Dialogs

Windows of varied types and styles facilitate the input and display of data. A *dialog* is a special type of window that has a two-way conversation with the application user. In contrast to the standard window, with its Action Bar in which the user must instruct the window what to do, dialogs engage in a conversation with the user, soliciting responses from the user and acting on them.

A dialog window, often called a *dialog box* or just *dialog*, contains a variety of control windows to facilitate the conversation with the application user. The Presentation Manager system provides a set of control windows for exactly this purpose:

- Radiobuttons
- Pushbuttons
- Check boxes
- Listboxes
- Entry (edit) fields
- Combo boxes
- Scroll Bars

Controls were discussed briefly in Chapter 11. Although these controls are usually used within a dialog window, they may be used in a client window of an application or any window, for that matter.

Many of these controls within the dialog box have interrelated purposes. For example, a scrollable list of items may be presented for selection from a listbox, and there may be an entry field control in the dialog for the user to type in a selection if it does not exist in the list. As a matter of fact, the combo box control you will read about shortly is just that.

Pushbuttons enable the user to indicate when selections are complete and the application may process the data obtained from the dialog. There are many types of buttons that have become common in the use of dialogs. An example is a Cancel button. Users click on such a button to back out of an operation begun when the dialog was initiated.

Another example is setting options that are saved within the application. The user may at some time in the future wish to return these settings to their default state—to their state when the application was installed. A Defaults button is useful to restore these options without forcing the user to reinstall the whole program. The OS/2 Control Panel's color setting is a good example of this.

With a little imagination, the controls you create in dialogs will become a very powerful part of your applications.

Communicating with Dialog Controls

The dialog owns its controls, much the same way as the frame window owns controls. The control windows send notification messages to their owner window when a significant event takes place. If the owner wishes to query the control for information or change the state of the control, it may accomplish this by sending messages to the control. In dialogs, the dialog window is the owner of the controls.

When an application wishes to insert an item into a listbox, it sends an `LM_INSERTITEM` message. If the owner of a button wishes to disable a button for some period of time, it sends a `WM_ENABLE` message to the button with the values of the message set to disable the button. Likewise, when a listbox item is selected, a `WM_CONTROL` with the `LN_SELECT` code is sent to the dialog window. An example of disabling a button with `WinSendMessage` is shown here:

```
WinSendMessage(hwndTheButton,WM_ENABLE,  
                (MPARAM)FALSE,  
                (MPARAM)NULL));
```

Messages may be sent or posted to dialog controls with `WinSendMessage` and `WinPostMsg`. This requires you to have the window handle of the window you wish to communicate with. Because dialogs are designed to have many controls, a special API was created specifically to send a message to a child of a window.

WinSendDlgItemMsg is the equivalent of a WinSendMessage call with the window handle returned by WinWindowFromID as the first parameter. This function can be used to send a message to the child of a window whose handle you already have. It will be received by the parent and forwarded to the child using the window identifier specified. This call is not restricted to dialogs. If you wish to send a message to any window that is a child of a window whose handle you have, WinSendDlgItemMsg performs that function.

```
reply = WinSendDlgItemMsg(hwndDialog,    /* Window whose child message is sent to */
                          ID_DLGCONTROL, /* Child window identifier                */
                          LM_INSERTITEM, /* Message identifier                    */
                          (MPARAM)param1, /* Message parameter 1                  */
                          (MPARAM)param2); /* Message parameter 2                  */
```

This code shows how you send a message to the dialog control with the identifier ID_DLGCONTROL.

You will soon see that a dialog has a window procedure, called a *dialog procedure*, and its own special default procedure, WinDefDlgProc. This is the dialog's counterpart to WinDefWindowProc.

When a control has something important to say to its owner, it sends or posts a notification to the dialog window. Just as with any other window procedure, the dialog procedure may look for WM_CONTROL and WM_COMMAND messages and interpret them as necessary. Dialogs are nothing more than windows that own controls.

When a dialog box is created, the dialog procedure is sent a WM_INITDLG message, just like a WM_CREATE is sent to the window procedure of a window being created. The dialog procedure may perform some initialization processing at this time.

Communicating with a dialog window or any of its controls is no different from communicating with any other window in the system.

Modality

There are two primary types of dialogs that you can create: modal or modeless. The one you select depends on what type of interaction with the rest of the system you would like to permit the user during the dialog.

When a modeless dialog is initiated, the user is still free to interact with any other window or application in the system. In a modal dialog, the user is restricted in what other objects on the screen he or she may interact with while engaged in the dialog. Modal dialog boxes are implemented to enable the user to interact with the dialog only and to inhibit interaction with other windows, according to how the modality is implemented.

Modal dialogs are used at times when you may need to inhibit user interaction with either the application creating the dialog or, in some cases, the rest of the Presentation Manager objects while the dialog takes place. Usually, a modal dialog is used when an application cannot continue until it gets the information it needs from the user.

Modeless Dialogs

A modeless dialog does not inhibit a user from working with any other object on the screen while the dialog takes place. A useful implementation of a modeless dialog would be one in which the user needs to supply information for the application but may need the resources of another application to provide that data.

Another implementation occurs when the dialog is used to change how data is displayed in a window while giving the user the ability to change the data at the same time.

The following code fragment shows how to implement a modeless dialog box. Once the dialog is initiated, messages are sent to it in the same way as to any other window.

```

hwndDialog = WinLoadDlg(HWND_DESKTOP,      /* Parent window handle */
                        hwndOwner,          /* Owner window handle */
                        (PFNWP)MyDlgProc,   /* Pointer to the dialog procedure */
                        (HMODULE)NULL,      /* Module handle if template is in a DLL */
                        ID_DLGID,           /* Dialog window ID */
                        (PVOID)NULL);       /* Presentation parameters */

WinShowWindow(hwndDialog, TRUE); /* Show the new window */

```

Modal Dialogs

When a modal dialog begins, the application user is prevented from interacting with a specified set of objects on the screen. Depending on the type of modality specified, the user is prevented from using either all other objects on the screen or just the other windows of the application that initiated the dialog. The two types of modal dialog boxes are system modal and application modal.

System Modal Dialogs When you create a system modal dialog box, the user of your dialog is prevented from using all other windows in the Presentation Manager session until that dialog is dismissed. This includes other applications, icons, and other parts of the program that created the dialog. Until the dialog is dismissed, the user may interact with only the dialog and any controls it creates.

System modal dialogs are not often needed but are available for those occasions where you may need all other Presentation Manager user input to be inhibited while the dialog takes place. The code shown creates a system modal dialog box.

```

hwndDialog = WinLoadDlg(HWND_DESKTOP,      /* Parent window handle */
                        hwndOwner,          /* Owner window handle */

```


Modal dialogs are used at times when you may need to inhibit user interaction with either the application creating the dialog or, in some cases, the rest of the Presentation Manager objects while the dialog takes place. Usually, a modal dialog is used when an application cannot continue until it gets the information it needs from the user.

Modeless Dialogs

A modeless dialog does not inhibit a user from working with any other object on the screen while the dialog takes place. A useful implementation of a modeless dialog would be one in which the user needs to supply information for the application but may need the resources of another application to provide that data.

Another implementation occurs when the dialog is used to change how data is displayed in a window while giving the user the ability to change the data at the same time.

The following code fragment shows how to implement a modeless dialog box. Once the dialog is initiated, messages are sent to it in the same way as to any other window.

```

hwndDialog = WinLoadDlg(HWND_DESKTOP,      /* Parent window handle */
                        hwndOwner,          /* Owner window handle */
                        (PFNWP)MyDlgProc,    /* Pointer to the dialog procedure */
                        (HMODULE)NULL,       /* Module handle if template is in a DLL */
                        ID_DLGID,           /* Dialog window ID */
                        (PVOID)NULL);       /* Presentation parameters */

WinShowWindow(hwndDialog, TRUE); /* Show the new window */

```

Modal Dialogs

When a modal dialog begins, the application user is prevented from interacting with a specified set of objects on the screen. Depending on the type of modality specified, the user is prevented from using either all other objects on the screen or just the other windows of the application that initiated the dialog. The two types of modal dialog boxes are system modal and application modal.

System Modal Dialogs When you create a system modal dialog box, the user of your dialog is prevented from using all other windows in the Presentation Manager session until that dialog is dismissed. This includes other applications, icons, and other parts of the program that created the dialog. Until the dialog is dismissed, the user may interact with only the dialog and any controls it creates.

System modal dialogs are not often needed but are available for those occasions where you may need all other Presentation Manager user input to be inhibited while the dialog takes place. The code shown creates a system modal dialog box.

```

hwndDialog = WinLoadDlg(HWND_DESKTOP,      /* Parent window handle */
                        hwndOwner,          /* Owner window handle */

```

```

        (PFNWP)&MyDlgProc,      /* Pointer to the dialog procedure */
        (HMODULE)NULL,         /* Module handle if template is in a DLL */
        ID_DLGID,              /* Dialog window ID */
        (PVOID)NULL);          /* Presentation parameters */

retval = WinProcessDlg(hwndDialog); /* Process the dialog and place the value */
                                     /* returned by WinDismissDlg in retval */

```

The other way to create a modal dialog is with `WinDlgBox`, which is actually just a combination of the above two functions:

```

retval = WinDlgBox(HWND_DESKTOP, /* Parent window handle */
                  hwndOwner,      /* Owner window handle */
                  (PFNWP)&MyDlgProc, /* Pointer to dialog procedure */
                  (HMODULE)NULL, /* Module handle if template is in a DLL */
                  ID_MODALDLG,    /* Dialog window ID */
                  (PVOID)NULL);   /* Presentation parameters */

```

/* The value returned in `retval` is the value passed back from `WinDismissDlg` */

Application Modal Dialogs An application modal dialog is probably the most-often-used type of dialog window. An application modal dialog is designed to inhibit user interaction with other parts of the application that created the dialog, while still allowing the user to interact with other applications. By definition, an application modal dialog prevents user interaction with the owner of the dialog and all children of that owner. This is one area where the difference between parent-child relationships and owner-owned relationships becomes very important.

Be careful when you program an application modal dialog. It is very easy to make a slip and freeze your system while you develop this type of dialog. The rule of thumb is that an application modal dialog must *never* be both the child of and owned by the same window.

As a standard, application modal dialogs should be a child of the desktop window (`HWND_DESKTOP`) and have the top-level window (the frame) of its application as its owner. Because the parent of the dialog is not the frame, the dialog is not frozen by the definition of application modality. The frame is the owner. By definition, all windows not a descendant of the frame (having the frame as its `PARENT`) are active, whereas windows that are children of the frame are frozen.

The other aspect of specifying the frame as the owner and the desktop as the parent is how the dialog displays and behaves with respect to positioning on the screen.

Because all windows are clipped to their parent, an application modal dialog may be positioned anywhere on the screen as long as `HWND_DESKTOP` is the parent. This rule is not enforced but is recommended in most situations. Some circumstances may dictate that other parent/owner windows be used, but consider the following situations:

- Say, for example, that you make the frame both parent and owner of the dialog. Because an application modal dialog prevents interaction with the dialog's owner and all descendants of that owner, the whole application becomes frozen, *including the dialog!* Because the frame is the owner, it and all children of it are not usable while the dialog is in progress. All of the frame's controls and children are frozen. Well, the dialog was a child of the frame, so it too is frozen! Now you can't use the dialog, so you can't dismiss it to allow the application to continue. The program is now useless.
- Almost the same holds true if the client of the top-level frame is specified as both the parent and the owner window of the dialog. The frame and its controls respond, but the client of the program and all its children (including the dialog) are useless. This defeats application modality, in that the user can still interact with the window's controls, but the user cannot even dismiss the dialog.
- Another pitfall is making the client the owner of the dialog. In this case, the whole purpose of an application modal dialog is defeated, as the frame and all of its controls are still active.

As you can see, the key element in the behavior of application modal dialogs is the owner, and the key element in the positioning and display is the parent.

One other aspect of modal dialogs is how input messages are handled. The whole reason for a modal dialog box is to inhibit the user from interacting with another part of the system while the dialog takes place. You can infer that input messages can't be handled in the normal manner, because messages should not get to the various windows on the screen.

When a modal dialog begins, the routing of messages generated by user input is handled by the Presentation Manager mechanism rather than the normal message processing path. This is so the Presentation Manager may control what messages get through to the windows on the screen. Specifically, messages need to get to the dialog exclusively. Note that this only applies for user input messages. All other application-generated messages flow freely through the system.

Dialog Controls

The OS/2 Presentation Manager shell provides several classes of controls that may be used within windows to communicate with application users. Each type of control is geared toward a different type of user input.

The dialog window is the owner of its controls. In addition, the general practice is to make the controls children of the dialog as well. This is done for clipping as well as modality reasons.

All notification messages generated by these controls are sent to their owner window. This is how the control communicates with the dialog. When a significant action is taken on a control by the user, the system sends a notification message to the control's owner. This notification message contains a notification code, such as `BN_CLICKED`, and other relevant information, such as the control identifier.

The dialog procedure receives these messages much in the same way a window procedure gets messages, and the procedure handles the message accordingly.

OS/2 provides several types of controls to enable users to interact with the dialogs. They are

- static selection fields. These are
 - a. radiobuttons for selecting a single item from a fixed list.
 - b. pushbuttons to cause the dialog to take immediate action.
 - c. check boxes for selection of multiple items from a fixed list.
- scrollable list of selectable items in a listbox.
- area where the user may type data, as in an entry (edit) field.
- control used to scroll areas within the dialog, scroll bars.
- combination control for selection: listbox and entry field together called a combo box.

Each of these controls may be placed anywhere, in any combination within the dialog window. We will discuss each of these controls and their uses.

Radiobuttons

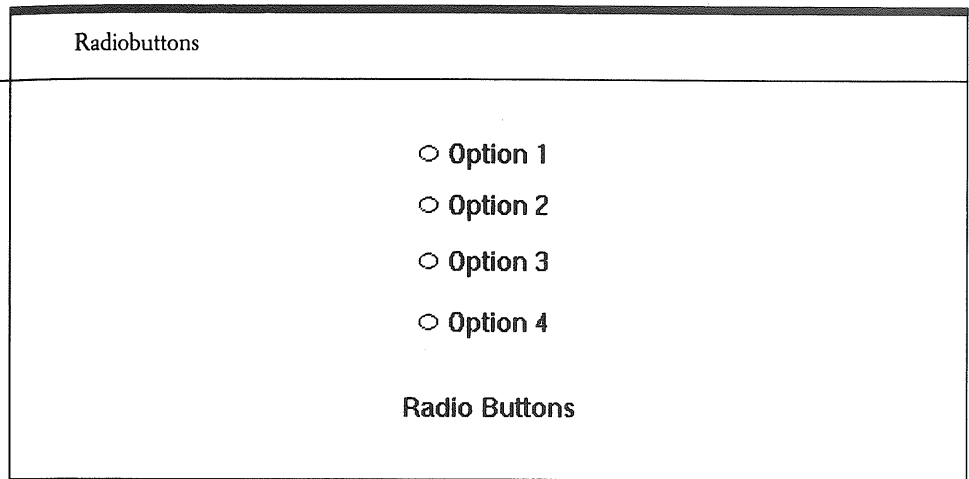
Radiobuttons present the user with a fixed list of choices of which only one at a time may be selected. The user selects from a group of buttons, such as the ones shown in Figure 13.1.

The system-defined radiobutton is presented as a circle next to a text string. The text is defined in the resource file and describes the selection the button represents. The circle indicates the selection state. A filled circle indicates that the button is selected; an empty circle indicates the button is deselected.

The system-defined class for the radiobutton (which is the same for *all* buttons) is `WC_BUTTON`. There are two styles available for radiobuttons within the `WC_BUTTON` class: `BS_RADIOBUTTON` and `BS_AUTORADIOBUTTON`. The difference between these two styles is what the application must handle when the user selects a button.

When the user selects a radiobutton, the screen shows that the selected button is highlighted; the previously selected button now displays as deselected (its circle is

Figure 13.1



empty). What actually occurs when a button is selected is the generation of a `WM_CONTROL` message, which is sent to the owner of the control. The `WM_CONTROL` message contains the identifier of the notification, in this case `BN_CLICKED`. This information is stored in the first message parameter of the message, along with the ID of the button that was clicked.

When you create buttons with the `BS_RADIOBUTTON` style, the dialog owning the buttons is responsible for highlighting and clearing the buttons on the screen. When a `BN_CLICKED` notification is received, it is the owner's responsibility to send a `BM_SETCHECK` message to the affected buttons. If the button selected is not previously in the selected state, the first message parameter (`MPARAM1`) of the `BM_SETCHECK` message must be set to `TRUE` to tell the button to change to the selected state. If it is already in this state, it is being deselected by the user and must be sent the `BM_SETCHECK` with `MPARAM1` set to `FALSE`. The owner must then send a message to the button that was previously in the selected state (if there was one) with `MPARAM1` set to `FALSE`. This clears the previously selected button.

If the style of the radiobuttons is `BS_AUTORADIOBUTTON`, the owner of them may ignore the `WM_CONTROL` message containing the `BN_CLICKED` identifier. With `BS_AUTORADIOBUTTON`, the system-defined procedures set the buttons' states when they are selected. The buttons are automatically set to the selected state if not previously set that way, and the buttons are deselected if they were in the selected state when another button was clicked.

For most implementations, autoradiobuttons are fine. For more detailed control and synchronization of events in conjunction with button selection, the `BS_RADIOBUTTON` style may be better for you.

When a group of buttons is created, one becomes the default. This button is initially displayed highlighted or selected. The setting may change throughout the course of

the dialog. By default, the first button in a group is initially displayed in this state. You can change the default in the `WM_INITDLG` processing when the dialog is created:

```
WM_INITDLG:
    WinSendMsg(hwndBut1,
        BM_SETHILITE,          /* Unset button 1 */
        (MPFROMSHORT)FALSE,
        NULL);
    WinSendMsg(hwndBut2,
        BM_SETHILITE,          /* Set button 2 */
        (MPFROMSHORT)TRUE,
        NULL);
    return((MRESULT)TRUE);
break;
```

This statement holds true for any of the controls being discussed. The first occurrence in a group is the default unless specified otherwise. It is important to remember to use `return((MRESULT)TRUE)` here to indicate to `WinDefDlgProc` that you have performed initialization processing on the dialog.

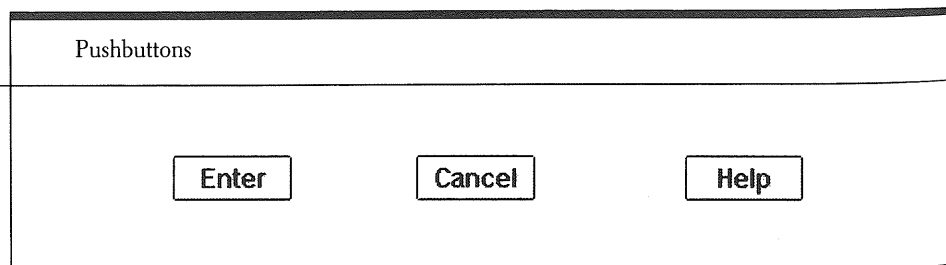
Pushbuttons

Pushbuttons are another instance of the `WC_BUTTON` class. This type of button is used to cause the application to act immediately when clicked upon. Common implementations of pushbuttons are buttons that cause the acceptance of data input during the dialog, buttons to dismiss the dialog and discard the input (a Cancel button), or ones that cause other windows to appear.

The style that is used for pushbuttons is `BS_PUSHBUTTON`. This type of control is displayed as a text string surrounded by a rectangle. Figure 13.2 shows this type of control.

When you click a pushbutton, the window that owns the button receives a `WM_COMMAND` message. The source identifier contained in the message is `CMDSRC_PUSHBUTTON` to distinguish it from a `WM_COMMAND` that may come from an accelerator or menu control. In addition, the message contains the command value of the button that was selected.

Figure 13.2



```
case WM_COMMAND:
```

```
    switch(SHORT1FROMMP(mp2))
    {
        case CMDSRC_ACCELERATOR:
            Look for the accelerators I am interested in
            and react to them accordingly
            break;

        case CMDSRC_PUSHBUTTON:
            Look for the button actions I am interested in
            and react accordingly
            break;

        case CMDSRC_MENU:
            Look for the menu options of interest
            and react accordingly
            break;
    }
    break;
```

This code shows how the parameters may be checked for WM_COMMAND to detect a menu selection, accelerator, or pushbutton selection.

Pushbuttons are usually placed in a dialog window, along with controls for user input such as check boxes and entry fields. The pushbuttons mainly enable the user to tell the application to process input, to back out of the dialog, or to bring up extended choices in another dialog.

When the user clicks on the Enter button, for example, the application should query the dialog's controls about their current state to record the user's selections. When a user clicks on Cancel, the dialog should simply be ended and the user's input discarded.

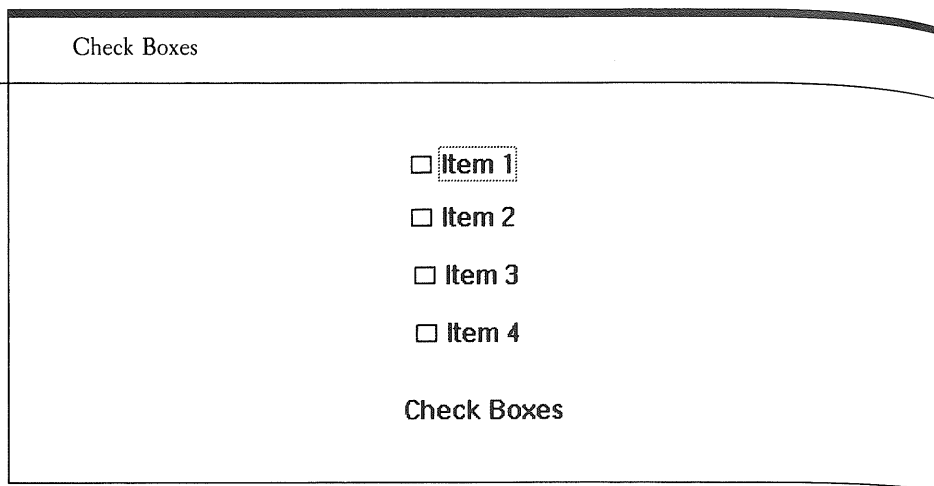
Check Boxes

Check boxes are a set of controls that allow your applications to present a fixed list of choices and enable users to select one or more items from the list. A group of check boxes represents a list of items that may be selected in any combination.

The system-defined check box is presented as a small square followed by a line of text that you enter to describe the choice the box represents. A selected check box is displayed with an X in the box; an unselected check box is an empty box, like those shown in Figure 13.3.

Although the name doesn't indicate it, a check box is also a button and belongs to the WC_BUTTON class. Its style is either BS_CHECKBOX or BS_AUTOCHECKBOX. Just like the radiobutton styles, both the BS_CHECKBOX and the BS_AUTOCHECKBOX receive a WM_CONTROL message with the BN_CLICKED identifier when the user clicks one.

Figure 13.3



Depending on the style, either the owner window or the system is responsible for painting the button in either its selected or deselected states. Analogous to the radiobuttons, the owner of the check boxes is responsible for sending `BM_SETCHECKs` to the `BS_CHECKBOXes`, but the system handles changing the state of `BS_AUTOCHECKBOXes`. Unlike the radiobuttons, however, when a user clicks one check box, others do not need to be deselected. That feature is what differentiates the two types of buttons. Check boxes are used for multiple selections, whereas radiobuttons are generally used for a single selection.

Listboxes

A listbox is a control that is used to present a dynamic list of selectable items to the user. As opposed to the selection buttons discussed so far that each represent a single selection, a listbox contains a *list* of items. In addition, the items in the listbox may change during the course of the application or even during the dialog. The other flexible feature of listboxes is that the list may be much larger than can physically fit in the box.

When there are more items than the box can physically hold, the listbox presents a scroll bar that can be used to scroll the items in the list until the desired entry is visible. The slider on the scroll bar shows which portion of the list is currently visible.

The reason these listboxes contain “dynamic” lists is that when you create a listbox, it is initially empty. In contrast to button controls, whose text and definition usually are defined in a resource file, a listbox is created as the need arises and is filled with the items available for selection.

During the course of the dialog, users may bring up a listbox for a selection, the details of which are not known until other selections are made. For example, picture

a dialog that manages files for your application. This dialog may have an Open button. Clicking on this button may produce a listbox of files from which the user selects a file to open. This list cannot be predefined, because the list of files may change at any given moment. In this case, your dialog reads from the current directory and places the contents into the listbox. This is a prime case of when a static text control such as a button is not adequate.

A listbox belongs to the system-defined `WC_LISTBOX` class. Several styles are available within this class. Each style gives the listbox certain capabilities and behavioral characteristics. These styles are

<code>LS_OWNERDRAW</code>	This style allows the listbox items to be drawn by the owner of the control.
<code>LS_MULTIPLESEL</code>	This allows multiple items to be selected from the listbox. Ordinarily, single selection is the default.
<code>LS_NOADJUSTPOS</code>	A listbox with this style is not automatically positioned and sized when its owner is. This style is used to finely tune the positioning of the listbox.

Most listboxes either take the default style or use `LS_MULTIPLESEL`. Ownerdraw listboxes are usually used to take over the actual drawing of the items into the control. For example, this is useful when the items you wish to display in the listbox are not standard text. Ownerdraw listboxes can be used to have a listbox full of selectable icons, to use another font, and so on.

When an `LS_OWNERDRAW` listbox is used, the owner window is responsible for setting the values for the listed items and for drawing each item in the listbox. Part of that responsibility is redrawing the items in their highlighted or normal states when necessary. This is used to draw custom items in the listbox that the usual "standard text string" can't handle.

Creating a Listbox A listbox is really just a window belonging to the `WC_LISTBOX` class. There are two ways to create a listbox. The first is to allow the system to create one automatically as a result of loading a dialog template that contains a listbox. The other method requires a call to `WinCreateWindow`. The code fragment shown here is an example of listbox creation using this second method.

```
hWndLbox = WinCreateWindow(hWndFrame,      /* The frame is the parent */
                           WC_LISTBOX,      /* The class is for a listbox */
                           NULL,            /* No window text for this sample */
                           LS_OWNERDRAW,    /* We want this to be ownerdraw */
                           0,0,0,0,         /* Initially invisible and 0 size */
                           0,0,0,0,0,0,0,0)
```

```

hwndFrame,    /* The frame is the owner too    */
HWND_TOP,     /* Place it on top of siblings    */
ID_LBOX,      /* The identifier is ID_LBOX      */
NULL,         /* No special control data flags  */
NULL);        /* Reserved field. NULL          */

```

Once this listbox is created, it can be filled with items using the `LM_INSERTITEM` message. `WinSetWindowPos` can then be called to make it visible.

Listboxes are empty until something is put into them. It is the listbox owner's responsibility to insert, remove, and obtain the status of the items in the listbox. These functions are accomplished by the `LM_*` messages. The message parameters contain the data specific to the message's function.

All the functions an owner window must perform on its listbox are accomplished through listbox messages. These messages are explained here:

Listbox Message	Effect
<code>LM_DELETEALL</code>	This message deletes all items in the listbox.
<code>LM_DELETEITEM</code>	This removes a single item from the listbox.
<code>LM_INSERTITEM</code>	This inserts an item at the specified location in the listbox. Several parameters can cause the item to be inserted at various places in the control.
<code>LIT_END</code>	This inserts the item at the end.
<code>LIT_SORTASCENDING</code>	This inserts the item in ascending order.
<code>LIT_SORT- DESCENDING</code>	This inserts the item in descending order.
<code>LM_QUERYITEM- COUNT</code>	Returns the number of items in the listbox.
<code>LM_QUERYITEM- HANDLE</code>	This message returns the handle of the item you provide the index for.
<code>LM_QUERYITEMTEXT</code>	This returns the text for a specified item.
<code>LM_QUERYITEMTEXT- LENGTH</code>	This returns the length of the text of a specified listbox item.
<code>LM_QUERY- SELECTION</code>	This message returns the index offset of a selected item.

LM_QUERYTOPINDEX	This returns the index of the item currently at the top of the listbox. This is important when you scroll listbox contents.
LM_SEARCHSTRING	This searches the listbox for a text string within the control and returns its index.
LM_SELECTITEM	This selects or deselects the specified item.
LM_SETITEMHANDLE	This message sets a handle to a specified item.
LM_SETITEMHEIGHT	This sets the height of the specified item.
LM_ITEMTEXT	This message sets the text of an item to the text in a buffer passed to the control.
LM_SETTOPINDEX	This scrolls a specified item to the top of the listbox.

Adding and deleting items from the listbox is accomplished by using the messages defined. Using `WinSendMsg` (or `WinSendDlgItemMsg`), you can control the behavior and display of the listbox quite easily. The example here demonstrates adding an item at the end of the listbox.

```
WinSendMsg(hwndLbox,          /* Listbox window handle */
            LM_INSERTITEM,     /* Message identifier */
            (MPFROMSHORT)LIT_END, /* Insert it at the end */
            (MPARAM)(PSZ)"Listbox Item"); /* Item to be inserted */
```

In general, as soon as a listbox is created, the program should loop through all the items that are to be displayed in the listbox and send `LM_INSERTITEM` to insert each one. Depending on how many items you wish to display, you may want to create the listbox initially invisible and execute `WinShowWindow` once all the items have been added. Unless there are only a few items, this is a good habit to get into. It eliminates unnecessary flicker in the listbox as each item is added.

Using a Listbox Once the listbox contains the items you want, the user may use the mouse or a combination of the cursor and Enter keys to select the entries. When an entry is selected, it becomes highlighted. Conversely, when an item is highlighted and then clicked upon, it becomes deselected. The same happens to an item highlighted in a single selection listbox when another item is highlighted. Whose responsibility it is to do this work depends on whether the `LS_OWNERDRAW` style has been specified or not.

The owner of a listbox without the `LS_OWNERDRAW` style need not worry about doing this work. This kind of listbox has all of the drawing of its items handled by the Presentation Manager system.

A feature of listboxes is that each item in the control has a handle available for use. This handle is not initialized when the listbox is created; it is there for program optimization.

You may store indexes (or any information) in the handle. For example, the use of this handle can speed searches through large lists. The `LM_SETITEMHANDLE` message gives a specified item a handle. The other message that makes listbox item handles effective is `LM_QUERYITEMHANDLE`. This returns the index of the item in the listbox that has the specified handle. This may be used to store other types of information as well as to speed up searches.

When a user clicks an item in the listbox, a `WM_CONTROL` notification message is sent to the owner window. The message contains the `LN_SELECT` notification code to inform the window that an item has been selected.

The other notification codes are sent as a result of other actions. The `LN_ENTER` notification is sent to the owner of the listbox when the user either has double clicked an item or has pressed the Enter key while the item was highlighted. The other notifications the listbox owner may receive are `LN_KILLFOCUS` and `LN_SETFOCUS`, which inform the owner that the listbox is losing or gaining the focus, respectively, and `LN_SCROLL`, which informs the owner that the listbox is about to scroll.

A listbox gives you the flexibility to change selection items dynamically, to scroll through the list should there be too many items to see at once, and to select multiple items. You can create listboxes as part of a dialog resource or dynamically, using `WinCreateWindow`. Using this control, you may give your users very large lists of choices in a very small space.

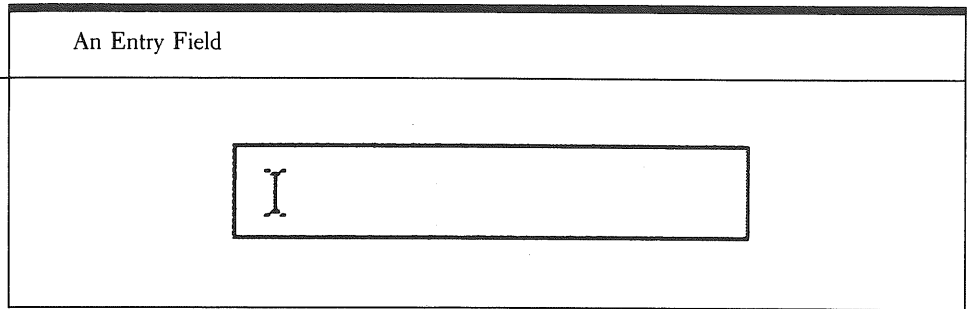
Entry Fields

An entry field (sometimes called an *edit field*) is a window in which the user may type text data. Only one line of text is allowed in an entry field; however, that line of text may be considerably longer than can fit within the window on the screen.

Entry fields solicit input from the application user. Buttons, listboxes, and the like have choices that the user may select. The entry field is an area in which the user may type data that is either not offered as a choice in any of the other controls or is needed to create new items. Figure 13.4 shows an example of an entry field.

An entry field belongs to the class defined by `WC_ENTRYFIELD`. The window really does not have any behavioral characteristics to speak of; it is a relatively plain control. This system-defined class has several styles that control how the data is displayed in the window. These styles are

Figure 13.4

**ES_MARGIN**

This is a frame with a margin drawn around the text.

ES_LEFT

The text in the window is left justified. This is the default unless overridden by one of the following styles.

ES_RIGHT

The text is right justified.

ES_CENTER

The text typed into the window is centered.

ES_READONLY

The text may not be changed.

ES_PICTUREMASK

Only specified parts of the entry field may be changed.

In addition, some styles enable you to manipulate both single- and double-byte characters. When a user wishes to enter text into an entry field window, the user gives it the focus by clicking it. This causes an I-beam cursor to appear in the window marking where the typed text will appear. Once the window has the focus, the user may type in the text. Text entry may be done in either insert or replace mode. This can be toggled by the user on a per-window basis.

Just like all the other types of controls, entry fields send notification messages to their owner window when significant events occur. An entry field sends a **WM_CONTROL** message to its owner when the user causes the text to scroll, when the user changes the text in some way, or when the control gains or loses the focus. The notification codes for entry-field-generated **WM_CONTROL** messages are

EN_CHANGE

The text in the entry field has changed.

EN_KILLFOCUS

The entry field is losing the input focus.

EN_SETFOCUS

The entry field is about to receive the focus.

EN_SCROLL

The entry field is about to scroll.

EN_MEMERROR	The system cannot allocate enough memory to hold a string of the size indicated in the EM_SETTEXTLIMIT message.
EN_OVERFLOW	The text length exceeds the size of the control.

In addition to the notification messages that come from the control, an entry field expects to receive certain messages from its owner window to instruct it how to behave in certain situations. The entry field manipulation messages are

EM_CLEAR	This message erases the selected text in the entry field.
EM_COPY	This copies the selected text into the clipboard. This is a nondestructive copy—the text remains unchanged in the entry field.
EM_CUT	This copies the selected text into the clipboard and removes it from the entry field.
EM_PASTE	This copies the text string in the clipboard to the entry field.
EM_QUERYCHANGED	The return value from this message tells whether the text in the entry field has changed.
EM_QUERY- FIRSTCHAR	This returns the offset of the first character in the entry field.
EM_QUERYREADONLY	This tells that the control is read only.
EM_QUERYSEL	This message returns the offset of the first selected character in the entry field.
EM_SETFIRSTCHAR	This sets the offset of the first displayed character in the entry field.
EM_SETINSERTMODE	This sets or resets insert mode.
EM_SETREADONLY	This sets or resets read-only mode.
EM_SETSEL	This sets both the beginning and ending offsets of the selected text in the control.

EM_SETTEXTLIMIT

This message is sent to the entry field, where it sets the maximum number of characters the entry field may contain. The default value is 32 and may be overridden by this message.

In addition to these messages, an application may query information about the entry field using the `WM_QUERYWINDOWPARAMS` message or the `WinQueryWindowText` API. They return various pieces of information about the entry field control.

Two new features introduced in OS/2 Version 1.2 are read-only entry fields and replace mode in entry fields. These features relieve constraints of entry fields that were included in Version 1.1. Now there is a built-in way for programs to protect data in these controls, so the programmer does not have to subclass the window to achieve the function.

Version 1.2 makes it possible to toggle insert and replace modes in edit controls, either under user control (with the Insert key) or using the `EM_SETINSERTMODE` message.

These notification and entry field control messages allow an application or dialog to monitor and modify the content of the entry field. Also built into the procedure that defines the entry field's behavior are functions that use the clipboard.

As is shown with the `EM_CUT`, `EM_COPY`, and `EM_PASTE` messages, an entry field has built-in clipboard interaction. Using a combination of keys while the entry field has the focus copies, cuts, or pastes text into the clipboard. Applications can cause this behavior directly by sending the `COPY`, `CUT`, or `PASTE` message, respectively.

Once the data in the entry field is to the application user's liking, the application usually provides an Enter button the user can click to accept the data. At this time, the owner may use messages to query the data in the control and pass it to the appropriate parts of the application.

Combo Boxes

Quite often it is desirable to combine the function of two controls so they act as one. One useful combination is the listbox and entry field. The operation of a combo box becomes clear if you picture a dialog to select a file. An initial entry or file specification (like *.DAT) is shown in the entry field, and a listbox of possible choices is displayed. Selecting an entry in the listbox replaces the old entry. This can then be selected or manually overtyped.

The class for combo boxes is `WC_COMBOBOX`. The available styles for the `WC_COMBOBOX` class are

CBS_SIMPLE	Both controls are always visible.
CBS_DROPDOWN	In addition to all of the characteristics of CBS_SIMPLE, the listbox is unusable until the prompt box is clicked.
CBS_DROPDOWNLIST	The entry field has static text that displays the current listbox selection.

The notification codes a combo box can send its owners are

CBN_EFCHANGE	The entry field has changed.
CBN_EFSCROLL	The entry field has scrolled.
CBN_MEMERROR	There is a memory allocation error.
CBN_LBSELECT	A listbox item has been selected.
CBN_LBSCROLL	The listbox has scrolled.
CBN_SHOWLIST	The listbox is being shown.
CBN_ENTER	The user has pressed Enter or double clicked on the listbox.

The set of messages that may be sent to a combo box to manipulate it are

CBM_SHOWLIST	This message shows or hides the listbox.
CBM_ISLISTSHOWING	This tells whether the listbox is visible.
CBM_HILITE	This changes the highlighting of a combo box prompt.
LM_SETCURSOR	This sets the cursor to a specific listbox item.
LM_QUERYCURSOR	This tells the position of the cursor.
LM_SETSELECTION	This message selects a specified item.
WM_UPDATESTYLE	This changes the style of the window.

Scroll Bars

Scroll bars are prevalent throughout the Presentation Manager interface. You can place a scroll bar anywhere within a window. An important fact to remember is that a scroll bar does not scroll a window or a portion of one by itself. Using the system may lead you to believe this, but the implementation of the scroll bar is what causes the scrolling behavior.

Like all of the other controls, scroll bars send notifications to their owner windows when important events occur. The events that affect operation of a scroll bar are the user clicking on the up, down, left, and right arrows; clicking and dragging the slider; or using accelerator keys such as PgUp and the arrow keys. When the scroll bar detects one or more of these events, it notifies its owner; the owner causes the window's contents to scroll.

Two types of scroll bars are available: vertical and horizontal. They are essentially the same, except that where a horizontal scroll bar may send a notification to scroll left, a vertical one may send a message to scroll up. In essence, these two types of scroll bars act in the same way.

The two types of scroll bar notification messages an owner may receive are `WM_VSCROLL` and `WM_HSCROLL`. These indicate that some action has been taken on a vertical scroll bar or a horizontal scroll bar, respectively. There is a set of notification codes within these messages that are common to both types of scroll bars. These common codes are

<code>SB_SLIDERPOSITION</code>	This tells the final slider position within the scroll bar when the user has finished scrolling.
<code>SB_SLIDERTRACK</code>	This informs the owner that the user is moving the slider within the scroll bar.
<code>SB_ENDSCROLL</code>	This indicates that the user has completed scrolling but was not using the slider.

There is also a set of notification codes that are unique to the type of scroll bar you are dealing with. Each of these messages has a counterpart for the other type of scroll bar. They are both outlined for completeness.

<code>SB_LINELEFT</code>	This code is sent when the user has either clicked on the left arrow of the scroll bar or pressed the left arrow key.
<code>SB_LINERIGHT</code>	This code is sent when the user has either clicked on the right arrow of the scroll bar or pressed the right arrow key.
<code>SB_PAGELEFT</code>	This code is sent when the user has either clicked on the area left of the slider or pressed the "page left" accelerator key.
<code>SB_PAGERIGHT</code>	This code is sent when the user has either clicked on the area right of the slider or has pressed the "page right" accelerator key.

The preceding are the notification codes specific to the horizontal scroll bar.

SB_LINEUP	This code is sent when the user has either clicked on the up arrow of the scroll bar or pressed the up arrow key.
SB_LINEDOWN	This code is sent when the user has either clicked on the down arrow of the scroll bar or pressed the down arrow key.
SB_PAGEUP	This code is sent when the user has either clicked on the area above the slider or has pressed the PgUp key.
SB_PAGEDOWN	This code is sent when the user has either clicked on the area below the slider or has pressed the PgDn key.

The preceding are the notification codes specific to the vertical scroll bar.

Be sure to differentiate here between scrolling as a result of keystrokes and mouse interaction with the scroll bar windows. When the application user scrolls via the mouse, the expected WM_VSCROLL and WM_HSCROLL messages are received as you would expect. However, when the user scrolls by pressing a key such as PgUp or linedown (down arrow), WM_xSCROLL messages are not sent to the window. These keys are treated just as any other keys. It is the window procedure's responsibility to post the appropriate WM_xSCROLL message with the proper parameters to the window. This simulates the message being posted as a result of the mouse click. In addition, the scroll bar slider position must be adjusted by the application to reflect the new position as a result of the scrolling. An example of how this may be accomplished is shown in Listing 13.1.

Listing 13.1

case WM_CHAR:

```

/*****
/* When the cursor keys are pressed, simulate the scroll messages as if */
/* they came from the scroll bar. Then post a message to the proper */
/* scroll bar to set the slider position after calculating where it */
/* should be. The variable "newpos" is a value that lies within the */
/* range of the scroll bar. The method used to calculate it is up to */
/* the application. */
*****/

if ( SHORT2FROMMP(mp2) == VK_UP) /* If the up arrow was pressed */
{
    if (SHORT1FROMMP(mp1) != KC_KEYUP) /* and it was key down (not = up) */
    {

```

```

        WinPostMsg(hwnd,
                    WM_VSCROLL,          /* send myself the scroll message */
                    hwndScrollBar,
                    (MPFROM2SHORT)FALSE,
                    SB_LINEUP);

        WinPostMsg(hwndSBar,           /* Calculate where slider should */
                    SBM_SETPOS,         /* go and put in "newpos" */
                    (MPFROMSHORT)newpos,
                    NULL);

        return((MRESULT)TRUE);
    }
}

if ( SHORT2FROMMP(mp2) == VK_DOWN) /* If the down arrow was pressed */
{
    if (SHORT1FROMMP(mp1) != KC_KEYUP) /* and it was key down (not = up) */
    {
        WinPostMsg(hwnd,
                    WM_VSCROLL,          /* send myself the scroll message */
                    hwndScrollBar,
                    (MPFROM2SHORT)FALSE,
                    SB_LINEDOWN);

        WinPostMsg(hwndSBar,           /* Calculate where slider should */
                    SBM_SETPOS,         /* go and put in "newpos" */
                    (MPFROMSHORT)newpos,
                    NULL);

        return((MRESULT)TRUE);
    }
}

if ( SHORT2FROMMP(mp2) == VK_PAGEUP) /* If the page up was pressed */
{
    if (SHORT1FROMMP(mp1) != KC_KEYUP) /* and it was key down (not = up) */
    {
        WinPostMsg(hwnd,
                    WM_VSCROLL,          /* send myself the scroll message */
                    hwndScrollBar,
                    (MPFROM2SHORT)FALSE,
                    SB_PAGEUP);

        WinPostMsg(hwndSBar,           /* Calculate where slider should */
                    SBM_SETPOS,         /* go and put in "newpos" */
                    (MPFROMSHORT)newpos,
                    NULL);

        return((MRESULT)TRUE);
    }
}

```

```

    }

    if ( SHORT2FROMMP(mp2) == VK_PAGEDOWN) /* If the page down was pressed */
    {
        if (SHORT1FROMMP(mp1) != KC_KEYUP) /* and it was key down (not = up) */
        {
            WinPostMsg(hwnd,
                        WM_VSCROLL,          /* send myself the scroll message */
                        hwndScrollBar,
                        (MPFROM2SHORT)FALSE,
                        SB_PAGEDOWN);

            WinPostMsg(hwndSBar,            /* Calculate where slider should */
                        SBM_SETPOS,          /* go and put in "newpos" */
                        (MPFROMSHORT)newpos,
                        NULL);

            return((MRESULT)TRUE);
        }
    }

    if ( SHORT2FROMMP(mp2) == VK_RIGHT) /* If the right arrow was pressed */
    {
        if (SHORT1FROMMP(mp1) != KC_KEYUP) /* and it was key down (not = up) */
        {
            WinPostMsg(hwnd,
                        WM_HSCROLL,          /* send myself the scroll message */
                        hwndScrollBar,
                        (MPFROM2SHORT)FALSE,
                        SB_LINERIGHT);

            WinPostMsg(hwndSBar,            /* Calculate where slider should */
                        SBM_SETPOS,          /* go and put in "newpos" */
                        (MPFROMSHORT)newpos,
                        NULL);

            return((MRESULT)TRUE);
        }
    }

    if ( SHORT2FROMMP(mp2) == VK_LEFT) /* If the left arrow was pressed */
    {
        if (SHORT1FROMMP(mp1) != KC_KEYUP) /* and it was key down (not = up) */
        {
            WinPostMsg(hwnd,
                        WM_HSCROLL,          /* send myself the scroll message */
                        hwndScrollBar,
                        (MPFROM2SHORT)FALSE,
                        SB_LINELEFT);

            WinPostMsg(hwndSBar,            /* Calculate where slider should */

```

```

        SBM_SETPOS,          /* go and put in "newpos"      */
        (MPFROMSHORT)newpos,
        NULL);

    return((MRESULT)TRUE);
}
}

break;

```

This code shows how to implement scrolling via the keyboard. Note that the `VK_*` values could have been "cased" to simplify the code even further.

The action your application takes as a result of the different notification codes depends on the items to be scrolled. In responding to an `SB_LINEUP` code within the `WM_VSCROLL` message, your application should scroll the smallest increment your display may allow. For example, if there is text to be scrolled, the action taken as a result of the `SB_LINEUP` should be to scroll one line of text. Depending on what type of data is being displayed, the `SB_LINE*` notification codes should be used to scroll in the smallest increment available, and the `SB_PAGE*` should cause a multi-increment scroll.

There are several ways to implement the actual scrolling. One way is to repaint the entire window. Another may be to use `WinScrollWindow`. This API scrolls the visible data, and you may then draw the new data in wherever it is appropriate. The other way you might wish to implement the scrolling is to change the presentation space's origin and then repaint it. This is often useful for high-performance scrolling on complicated graphics.

The scroll bar control messages that the owner may send to the control query various settings that the scroll bar currently has, or they may change its settings and can force scrolling. These `WM_CONTROL` values are

<code>SBM_QUERYHILITE</code>	This message returns the highlighting status of the scroll bar.
<code>SBM_QUERYPOS</code>	This message returns the position of the slider.
<code>SBM_QUERYRANGE</code>	This returns the scroll bar range of the currently displayed items.
<code>SBM_SETHILITE</code>	This message sets the scroll bar highlighting status.
<code>SBM_SETPOS</code>	This message sets the slider position on the scroll bar.
<code>SBM_SETSCROLLBAR</code>	This message sets the scroll bar range and slider position.

Scroll bars allow users to scroll the contents of windows. Through the specialized scroll messages, scrolling as fine as one pel at a time or as broad as pages at a time is possible. The amount of freedom for scrolling is bounded only by the application design.

Multiline Edit Controls

A *multiline edit control (MLE)* is just like an entry field control (edit control) that handles multiple lines. Each MLE control is a member of the `WC_MLE` class. Rather than having the edit control scroll when the right edge is reached, an MLE control wraps around to the next line.

A line break can also be made when the user wishes to break the line at desired points.

The styles available within the `WC_MLE` class are

<code>MLS_READONLY</code>	The control's contents may not be changed.
<code>MLS_BORDER</code>	The control has a border around it.
<code>MLS_WORDWRAP</code>	The wordwrap feature is on.
<code>MLS_HSCROLL</code>	The control has a horizontal scroll bar.
<code>MLS_VSCROLL</code>	The control has a vertical scroll bar.
<code>MLS_IGNORETAB</code>	The tab key is ignored by the control.

The notification codes that may be received by an MLE's owner within a `WM_CONTROL` message are

<code>MLN_TEXT-OVERFLOW</code>	The text limit has been reached.
<code>MLN_PIXHORZ-OVERFLOW</code>	The input exceeds the horizontal size of the control.
<code>MLN_PIXVERT-OVERFLOW</code>	The input exceeds the vertical size of the control.
<code>MLN_OVERFLOW</code>	The control has been changed so the existing text no longer fits.
<code>MLN_HSCROLL</code>	The control is about to scroll horizontally.
<code>MLN_VSCROLL</code>	The control is about to scroll vertically.
<code>MLN_CHANGE</code>	The text has changed.
<code>MLN_UNDO-OVERFLOW</code>	The undo limit has been reached so UNDO cannot be done.

MLN_CLIPBDFAIL	A clipboard operation has failed.
MLN_MEMERROR	A memory error has occurred.
MLN_SETFOCUS	The control has received the focus.
MLN_KILLFOCUS	The control has lost the focus.
MLN_MARGIN	The mouse has entered the window.
MLN_SEARCHPAUSE	A search is being performed.

A large number of messages may be sent to an MLE control. There are messages ranging from setting the wordwrap mode, to setting and querying data formats, to cutting and pasting to the clipboard. A complete listing is given in the *IBM OS/2 Programming Guide*.

Dialog Procedure

When an action is taken in the course of an application that displays a dialog window, either `WinLoadDlg` or `WinDlgBox` is called. You use these APIs to create the dialog box. Depending on the method you use for creation and modality, the dialog window receives the input focus, and user input messages are directed to that window.

A dialog window is really nothing more than a window with its own type of default processing. The dialog procedure is just another window procedure that is designed to handle a different set of messages than a regular, run-of-the-mill client window procedure. The structure of a dialog procedure is the same as a window procedure, except that the messages it expects to receive require a different process than a "normal" window procedure.

Because the dialog procedure is the owner of potentially many controls within the dialog, it receives many notification messages from them. Each notification means something different to each particular dialog, and each may react differently to a notification. For example, a dialog with a group of pushbuttons may respond to a `WM_CONTROL` with the `BN_CLICKED` notification differently than a dialog receiving the same notification for a group of autoradiobuttons.

The other departure from a "normal" window procedure is that for all messages that the dialog procedure does not wish to handle, it calls `WinDefDlgProc`. This is the default procedure for dialog windows. `WinDefDlgProc` is similar to `WinDefWindowProc` in that it is able to process any message a dialog window may expect to receive and may not wish to process.

A dialog procedure can usually expect to receive notifications from its controls, size, movement messages, and `WM_INITDLG`. This is not to say that other messages,

such as `WM_BUTTON1DOWN` messages, may not be sent to a dialog procedure. Rather, it is unusual for other messages to occur, given the nature of a dialog window.

Some of the common messages a dialog procedure must handle are

`WM_CHAR`

Just like any window procedure, this message indicates a keystroke. For the Enter key, if there is a button that is highlighted, this button is sent a `BM_CLICK` message. All keystrokes entered while the dialog has the focus send this message to the dialog.

`WM_INITDLG`

This message is sent to the dialog's dialog procedure as the dialog is being created. This gives the dialog procedure the chance to perform some initialization before the dialog is visible. This technique is often used to fill listboxes before they are displayed.

`WM_CONTROL`

This message is posted to the dialog by a control when a significant event occurs. The message may be handled exactly the same way as any other window handling this message. The parameters identify the source of the message and any other relevant information.

`WM_COMMAND`

This message is posted when a control, such as a pushbutton or menu, wishes to notify its owner of events. It may be processed by the dialog the same way as other windows.

The actions performed by a dialog procedure as a response to messages are similar to those performed by any window procedure. Along with similar actions go the same responsibilities that a window procedure has. Most notably, a dialog procedure must handle messages in an expeditious manner. The 1/10 second rule applies to dialog procedures as well as other window procedures.

The structure of the dialog procedure is the same as a window procedure. It accepts the window handle, message identifier, and the two message parameters as arguments and uses a `CASE` statement to filter the messages. At the end the dialog procedure passes all messages it is not interested in to `WinDefDlgProc`. To terminate a modal dialog, use the `WinDismissDlg` function call.

In a window procedure, a main window may terminate itself by posting a `WM_QUIT` message to itself. Other windows may be terminated as the result of a

WinDestroyWindow call. Dialogs are terminated when the user indicates that he or she is finished with the dialog. WinDismissDialog is used to terminate a modal dialog. If the dialog is modeless, its owner must call WinDestroyWindow to eliminate the dialog window. This function is usually executed as the result of receipt of a BN_CLICKED notification of an Enter or Cancel pushbutton. The code in Listing 13.2 shows a sample dialog procedure and how it processes messages and terminates itself when the user is done.

Listing 13.2

```
MRESULT EXPENTRY MyDlgProc( hwndDlg, message, mp1, mp2 )
HWND    hwndDlg;
USHORT  message;
MPARAM  mp1;
MPARAM  mp2;
{
    switch (message)
    {
        case WM_INITDLG:
            Perform default button processing;
            Initialize some data;
            return((MRESULT)TRUE);
            break;

        case WM_COMMAND:
            switch( SHORT1FROMMP( mp1 ) )
            {

                /*****
                /* If the dialog information is accepted by pressing the Enter */
                /* button or the Enter key, read the controls and act if      */
                /* necessary                                                  */
                *****/

                case DID_OK:
                    Query the control's information;
                    Perform any other actions necessary;
                    break;

                /*****
                /* If the dialog was canceled, dismiss the dialog without    */
                /* accepting any of the data                                  */
                *****/

                case DID_CANCEL:
                    WinDismissDlg(hwndDlg,TRUE);    /* Dismiss the dialog */
                    break;
            }
        default:
            return(WinDefDlgProc( hwndDlg, message, mp1, mp2 ));
    }
}
```

```
    }  
    return((MRESULT)TRUE);  
}
```

When Enter or Cancel is detected, the dialog procedure queries the status of the various controls it owns to determine what the user has selected. Once this is done, the procedure calls `WinDismissDlg`, which destroys the modal dialog and its controls.

Message Boxes

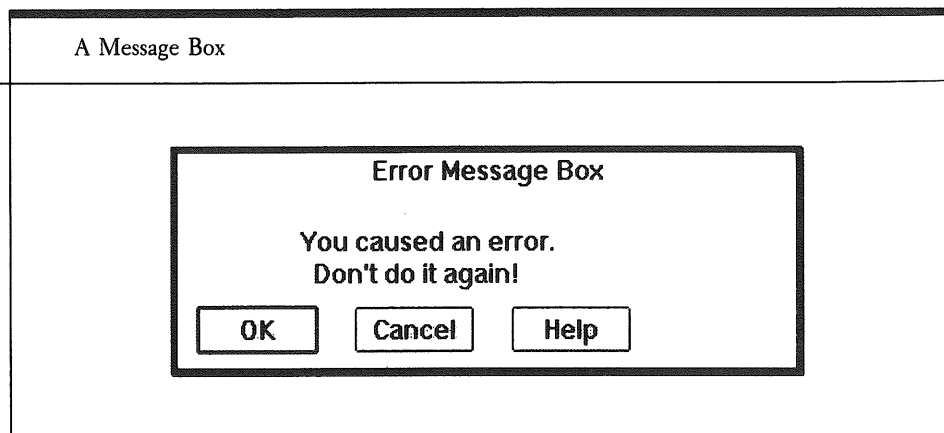
A message box is a modal dialog box that is used to present messages to the user. In many cases, message boxes are used to inform users when problems are encountered while the program attempts to perform a particular function. Other times they are used to ask the user for confirmation of an operation that is about to be performed. They may also be used to assist in debugging applications. Messages may be displayed to tell you what the state of your application is. Message boxes are simply stripped down, fast dialog boxes that offer a small set of user input functions.

Programming-wise, message boxes are just simplified modal dialog boxes that present the user with some text and one or more pushbuttons. These buttons enable the user to respond to the displayed messages. Figure 13.5 shows a sample message box.

A message box is not designed to be a full-fledged dialog. It therefore does not need a template or a dialog procedure. It is an easy-to-create and easy-to-use dialog that may be created simply and “on the fly.” The creation of a message box requires no more preparation than a call to `WinMessageBox`.

```
WinMessageBox(HWND_DESKTOP,          /* Parent window handle */  
              hwndMyFrame,           /* Owner window handle  */
```

Figure 13.5



```
(PSZ)"The function failed.\nSee the owner's manual", /* Text */  
"It Blew up.", /* Message box title text */  
MBID_MSGBOXID, /* Message box identifier */  
MB_OKCANCEL|MB_ICONEXCLAMATION); /* Message box window style */
```

When coding the `WinMessageBox` function call, you must remember that a message box is really just a modal dialog box. As such, it is subject to the same visibility and behavioral characteristics outlined in the discussion about modal dialog boxes.

The first two parameters to the `WinMessageBox` call are the parent and owner window handles. In general, `HWND_DESKTOP` should be the parent. The frame window handle for the window that caused the creation of the message box should be specified as the owner. This window will be made the active window when the message box goes away.

The next two parameters in the `WinMessageBox` call are the text string and title string. These are the two text strings that are displayed inside the message box. This is a static area that is 32 characters wide. Multiple lines may be displayed in the text portion of the window, however. This is demonstrated in the code sample just given. By using the C language *newline* character, `\n`, multiple lines of text may be displayed in the message box. This text is what is used to inform the user about whatever it is that the message box was created for.

The next item that must be specified is the message box's identifier. This is only necessary if a help function is being used. This value will be passed to the function in the event of a `WM_HELP` message as a result of the Help button selection. If the help function is not enabled for this message box, this value may be `NULL`.

The final item passed to `WinMessageBox` is a value that represents the styles that the message box will possess. The styles indicate what buttons the message box will display, how it will behave, and whether it will have icons. This may be done all in one value, because the function for the message box is not very complex and all of the style bits can be fit into one 16-bit word.

The styles available for message boxes are

- `MB_OK`
- `MB_OKCANCEL`
- `MB_CANCEL`
- `MB_ENTER`
- `MB_ENTERCANCEL`
- `MB_RETRYCANCEL`
- `MB_ABORTRETRYIGNORE`
- `MB_YESNO`

- MB_YESNOCANCEL
- MB_HELP

The preceding are all the types of buttons available for the message box.

- MB_NOICON
- MB_ICONHAND
- MB_ICONQUESTION
- MB_ICONEXCLAMATION
- MB_ICONASTERISK

The second list contains the identifiers for all of the available icons to be contained in the message box.

- MB_DEFBUTTON1
- MB_DEFBUTTON2
- MB_DEFBUTTON3

The preceding items identify which button in the message box to make the default.

- MB_APPMODAL
- MB_SYSTEMMODAL

The two items just listed are the indicators to specify the type of modality.

The items in these lists may be ORed together to specify a combination of styles for the message box. For example,

```
MB_ENTERCANCEL!MB_ICONEXCLAMATION!MB_DEFBUTTON2!MB_SYSTEMMODAL
```

creates a system modal message box with Enter and Cancel buttons, with the Cancel button as the default, and with an exclamation point icon. Note that if no modality is specified, MB_APPMODAL is assumed.

When the user selects one of the buttons in the message box, a WM_CONTROL notification is sent containing a value corresponding to the button selected. The values that a window may expect as a result of a selection are

- MBID_ABORT
- MBID_CANCEL
- MBID_ENTER
- MBID_ERROR

- MBID_IGNORE
- MBID_NO
- MBID_OK
- MBID_RETRY
- MBID_YES

Each of the notification codes corresponds to the type of button selected. Notice that there is no code for the Help button. This is due to the fact that a `WM_HELP` message will be sent to the owner of the message box if this button is selected.

Message boxes may be used to display a window of information to a user without going through the steps to create a complex window. The message box is supplied to give programmers a fast way to give the user information in case of some kind of system or application failure. They can handle only limited input; however, they are presented in such a way that the application may interpret the response and attempt a graceful termination of the offending process.

Message boxes have also been useful in debugging applications. Any time you would like to see some information during the course of an application, all you have to do is display a quick message box with the information you would like to see. This is similar to adding a `printf` to your code to print out debugging information.

Summary

Dialogs are the means applications use to communicate with their users. A dialog window is designed to contain a variety of controls to display selections for and solicit input from users. Every control has a way of telling the dialog window what happened and who caused the action. These notifications are how the dialog knows what the user is doing and give it the chance to respond.

The biggest difference between a dialog and any other window is that a dialog converses with the user. It asks the user for information as well as responds to the user's commands. Other windows generally do not perform this type of function. Instead, the other windows create dialog windows when this type of interaction is desired.

Dialogs and controls are used primarily to talk to the user, whereas other types of windows are used to display the data indicated by the dialog.

User Menus

Menus are the way a user initiates actions within an application. The way a Presentation Manager application presents menus to the user is with an Action Bar. As you saw in Section 2, the Action Bar is immediately below the Title Bar in a standard window. This Action Bar, which is another type of window, contains menu pull-downs that the user interacts with. This chapter discusses how to create, control, and communicate with menus.

Menus are also windows with their own attributes and behavior. The behavior is defined by the attributes the menus possess. Two types of items may be selected from a menu. The first is another menu (or *submenu*, as it is sometimes called). When the user selects a menu from a menu, a pull-down appears. This pull-down may contain other submenus. It may also contain menu items.

The second item that may be selected from a menu is an option or menu choice. When such an item is selected, a `WM_COMMAND` message is generated and sent to the owner window. Inside this message is the identifier of the particular menu item that was selected.

The two types of selections are very similar. A submenu window is just a menu item with the “submenu flag” set to `TRUE`. They are displayed in a similar way. The only real difference is what action each takes when selected.

The handling of menus is done primarily by the system. Application programs are not responsible for making the items on pull-downs appear, nor are they responsible for drawing the menus or menu items. What you do with menus in your programs is define the menus in the resource file or create menu templates in memory. They may

then be created as part of a standard window or generated during the course of program execution.

Menus are usually defined in the resource definition file in a menu template. When a standard window is created including the menu identifier from the resource file and the `FCF_MENU` control flag, an Action Bar is created as part of the window. Inside the call to `WinCreateStdWindow`, the resource definition for the menu is read and a template is created in memory for the menu. The Action Bar is then generated with the menus accordingly placed in the Action Bar.

The Action Bar has a system-defined identifier, `FID_MENU`. This ID may be used to obtain the handle of the Action Bar to work down to the menu window handles. A useful API for such functions is `WinWindowFromID`, as described in Chapter 11.

```
hwndActBar = WinWindowFromID(hwndFrame,FID_MENU);  
hwndMI1    = WinWindowFromID(hwndActBar,ID_Menu_1);
```

This code fragment demonstrates how to use the frame window handle to obtain the handle of one of its children, the Action Bar. Then, using that handle, you can get the handle of the menu item (or submenu) you wish to work with. That handle may then be used to send messages to disable the menu, add menu items to a submenu, or query the status of the menu.

When you go about designing an application, menus should be ordered into functional groups. In the example in Figure 14-1, you can see one way to group menus. The Files pull-down includes options to manipulate files and make files available for use. The Advanced pull-down can be used for functions such as the clipboard or DDE. Other common options for a main window are choices to configure the application; there is also usually an Exit submenu that contains options and instructions to save your current file or allow you to stop the exit procedure if you change your mind.

Many menu item selections bring up a dialog box to request additional information. A perfect example is the Files pull-down. If a user were to select its Open option, a dialog would be created to question the user about which file to open. Some conventions have developed for menus. One such convention is that if a menu item causes a dialog to appear when selected, that item's text is followed by an ellipsis. For example, the Open menu item displays as `Open...` on the menu.

Each of these functions is a response to the `WM_COMMAND` message generated by the menu control when a selection is made. This is how menus function and communicate. We will now explore how to make menus work for us.

Creating Menus

The Action Bar, its associated menu pull-downs, and its menu items are specified inside the resource definition file. When this file is compiled and placed in the executable file by the resource compiler, a menu template is created inside the executable module.


```
/* The FCF_MENU flag in this case is ORed into the myctldata variable and */
/* the associated resource identifier is IDM_THEMENU. */
```

```
#define IDM_THEMENU 100
#define IDMOPEN 110
#define IDMSAVE 120
#define IDMDELETE 130
#define IDMFONT 140
#define IDMCOLOR 150
#define IDMCUT 160
#define IDMCOPY 170
#define IDMPASTE 180
#define IDMEXIT 190
#define IDMDONTEXIT 200
#define IDMFILES 500
#define IDMOPTIONS 501
#define IDMCLIPB 502
#define IDMEXIT 503
#define IDMHLP 504
#define IDMOURSUBMENU 505

MENU IDM_THEMENU PRELOAD
BEGIN
    SUBMENU "Files", IDMOURSUBMENU
    BEGIN
        MENUITEM "Open...", IDMOPEN, MIS_TEXT
        MENUITEM "Save", IDMSAVE, MIS_TEXT
        MENUITEM "Delete", IDMDELETE, MIS_TEXT
    END
    SUBMENU "Options", IDMOPTIONS
    BEGIN
        MENUITEM "Font", IDMFONT, MIS_TEXT, MIA_CHECKED
        MENUITEM "Colors", IDMCOLOR, MIS_TEXT
    END
    SUBMENU "Clipboard", IDMCLIPB
    BEGIN
        MENUITEM "Cut", IDMCUT, MIS_TEXT, MIA_DISABLED
        MENUITEM "Copy", IDMCOPY, MIS_TEXT
        MENUITEM "Paste", IDMPASTE, MIS_TEXT
    END
    SUBMENU "Exit", IDMEXIT
    BEGIN
        MENUITEM "Exit Program", IDMEXIT, MIS_TEXT
        MENUITEM "Return to Program", IDMDONTEXIT, MIS_TEXT
    END
    MENUITEM "F1=Help", IDMHLP, MIS_HELP:MIS_BUTTONSEPARATOR
END
```

```
/* The sample menu resource definition */
```

The code just shown is a menu definition that may be placed in the resource definition file. This is the preferred method of menu creation. All menus and menu objects may be created “on the fly” with a sequence of API calls. This is an involved process and ties the menu definition into your source code. By using the resource definition file and separating the menu definition from the source code, you may modify the menu without necessarily having to rebuild the whole application. The only action the program needs to handle is responding to the menu. OS/2 creates the menu for you. During the discussion of communicating with menus, you will see how to modify menus and change attributes during program execution.

The first line in the preceding code tells that this is a menu definition and the system creates it with the behavior defined by the `WC_MENU` class. The identifier is nothing more than an integer. To enhance readability, mnemonics have been assigned to all of the resource identifiers. Placing `#define`’s into a private header file makes it easy to use the mnemonic in both the source (where the resource ID must match what is defined here) and in any resource definition.

The core of the menu definition is block structured, much the same as program source code. The `BEGIN` and `END` keywords are used for logical separation of function. The menu definition must start with `BEGIN` and finish with `END`.

The two statements used in the menu definition are `SUBMENU` and `MENUITEM`. As discussed earlier in this chapter, a submenu contains either menu items or other submenus. When the user selects a submenu, another pull-down appears; if the user selects a menu item, a `WM_COMMAND` is sent to your window with the identifier of the selection.

When Action Bar items are defined, one or more submenus may be defined within them. Although menu items may be placed on the Action Bar, the Common User Access (CUA) guidelines state that all items on the Action Bar will be pull-downs. The only deviation that is allowed is the help item. The CUA states that there may be no menu items on the Action Bar. Each of the pull-downs contains one or more submenus or menu items. The `SUBMENU` keyword is used to indicate that the item being defined causes a pull-down to appear. The text to be displayed for this Action Bar item is specified, along with a mnemonic identifier.

When the `SUBMENU` keyword is used, a block definition of the submenu is begun. This block is delimited with `BEGIN` and `END`. Inside the block may be more blocks of submenu definitions or menu items.

The `MENUITEM` statement defines the items displayed inside the menu. Following the `MENUITEM` keyword is the text to be displayed, the identifier that will be sent inside the `WM_COMMAND` message when the item is selected, and the combination of the `MIS_` and `MIA_` values that define the styles and attributes the item will possess. The following list explains all of these items. These styles and attributes may be ORed together to create items with several combined behaviors.

Style	Description
MIS_BITMAP	This object is displayed in the menu as a bitmap.
MIS_BREAK	This style indicates that this item is to be the last in a row or column and the next item is to be displayed at the beginning of the next row or column.
MIS_BREAK- SEPARATOR	This style may only be used in a SUB-MENU statement. It draws a separator between the columns or rows of the menu.
MIS_BUTTON- SEPARATOR	This item is a button. It is automatically shown with a separator. There may be up to two of these items in a menu, and they are the last items in the menu. In addition, the user may only select these items with the mouse or a menu accelerator.
MIS_HELP	When a menu item with this style is selected, a WM_HELP message is sent to the owner rather than the WM_COMMAND.
MIS_OWNERDRAW	This is similar to the owner draw style of listboxes. The owner of the menu must draw the item as a result of the WM_MEASUREITEM and WM_DRAWITEM notification messages.
MIS_SEPARATOR	This item is nothing more than a horizontal dividing line. It is not selectable and is used only to divide a set of choices to make the appearance of the menu flow well.
MIS_STATIC	This is a text item that is not selectable. It is displayed for information only.
MIS_SUBMENU	This item is a submenu. When the item having this style is selected, a pull-down appears. The SUBMENU keyword in the definition is really a MENUITEM with this style.

MIS_SYSCOMMAND	This indicates that the menu item should send a WM_SYSCOMMAND message rather than the WM_COMMAND that would be sent otherwise.
MIS_TEXT	The object displayed in the menu is a text string.
Attribute	Description
MIA_CHECKED	If the state of this attribute is TRUE, a check mark is displayed next to the item.
MIA_DISABLED	If this attribute is TRUE, the item is disabled and may not be selected. It is drawn in a "grayed" or dimmed state.
MIA_FRAMED	If this attribute is TRUE, a frame is drawn around the item.
MIA_HILITED	This attribute is TRUE if the item is selected.
MIA_NODISMISS	If a submenu has this attribute set to TRUE, its pull-down is not removed before the application's window receives the WM_COMMAND message.
MIA_SELECTED	When the item is selected, this attribute is set to TRUE.

A feature in the text specification of the menu items and submenus is the use of the ~ (tilde) character to specify a menu accelerator. The character preceded by a ~ is displayed with an underscore and is an accelerator when that pull-down or menu item is displayed. When the Alt key is pressed along with the character mnemonic for an item, the menu responds as if that item were selected. Take care, however, to ensure that you do not specify the same character for two different items that are both available at the same time.

Creating the menu requires nothing more than writing this definition, adding it to the resource file, and setting up the proper flags and values in your window creation calls. When the resources are compiled and linked with the program, the menus will be available for use when the windows using them are created.

Communicating with the Menu

Because all submenus and menu items are windows, the method of communicating with them is the same as any other window: messages. When actions occur that affect the menu, notification messages are sent to the owner window, in this case, the frame.

Menus are complex in their owner structure. The Action Bar window is the owner of the menus. When you are talking about the "Menu" for an application, you are really talking about the Action Bar. The "items" on the Action Bar are really menus. As you have seen, these menus may have other menus and/or items. Each of these elements is a window with unique identifiers, as specified in the definition.

Because each of these items is a window, each has its own window handle. The creation of windows is handled by the system when the Action Bar is created as part of the window. Each window handle may be obtained to work with each item separately by using its ID and the `WinWindowFromID` function.

Detecting Menu Selection

When an Action Bar item is selected, a pull-down appears. The system handles this action, and no program intervention is necessary. This holds true for any submenu. The operating system takes care of displaying pull-downs and also handles removing them when the user exits the menu. As such, no action need be taken when a menu is pulled down. When a menu item is selected, however, notifications are sent up the owner chain, eventually to get to the application's window procedure as a `WM_COMMAND` message.

When a menu item is selected, either via the mouse or keyboard, the system posts a message to the menu control. The window procedure for the control sends an `MM_SELECTITEM` message to the Action Bar, which in turn sends a `WM_MENUSELECT` message to the owner. In this example, the owner is the frame window. The default window procedure for the frame processes this message and sets the reply to it to be `TRUE`. This indicates to the control that no errors have occurred and it should proceed. The control then sends a `WM_COMMAND` message to the owner (frame) who will pass the message to the client window.

The client window is then responsible for processing the `WM_COMMAND`. Inside the `WM_COMMAND` is the identifier associated with the menu selection. This identifier is the one defined in the menu definition for that item. This identifier is contained in the first of the two message parameters. The following code section shows how you can incorporate the menu selection identification code into your window procedure.

```
MRESULT FAR PASCAL WindowProc( hWnd, message, lParam1, lParam2 )
HWND    hWnd;
USHORT  message;
```

USER MENUS

```

MPARAM lParam1;
MPARAM lParam2;
{
    /* As with any window procedure, we execute a switch statement with the */
    /* message we received as the target of the switch. All messages for */
    /* our window will come here and we will act on each. */
    switch (message)
    {
        case WM_PAINT:
            Execute painting routine.....
            break;

        /* If the message is a WM_COMMAND, execute a switch statement to */
        /* determine the identifier to tell us what action to take. */
        /* The identifier is contained in the first message parameter */
        case WM_COMMAND:
            switch ((USHORT)lParam1)
            {
                case IDMOPEN:          /* If the menu selection was "Open" */
                    Execute file open code
                    Set value and return
                    break;

                case IDMSAVE:          /* If the menu selection was "Save" */
                    Execute file save code
                    Set value and return
                    break;

                .
                .
                .

                case IDMFONT:          /* If the menu selection was "Fonts" */
                    Execute code for font manipulation
                    Set value and return
                    break;
            }
            break;          /* Finish switch for WM_COMMAND identifiers */

        default:
            Call the WinDefWindowProc
            break;
    } /* End switch looking for different messages */
}

```

As you can see, telling what menu item was selected is no more difficult than checking the parameters on any other type of message. Once it has been determined which item

has been selected by the user, the program may take the appropriate action. Because the system takes care of pulling down menus and highlighting selected items, all your program needs to do is to respond to a `WM_COMMAND` message when an item is selected.

Controlling the Menu

As with the dialog control windows, menu control windows have a set of messages defined that they will respond to. These are the `MM_*` messages. There are messages to query and set the state of menu items, messages to insert and remove items, and start and end menus under program rather than user control. With these system-defined messages, you may modify existing menus according to choices being selected by the user. Using these messages, entire menus may be built.

Each menu item is a window and has a window handle so you can send messages directly to the window. Additionally, there are many menu messages that have a flag to search a submenu. What this says is, "When you get this message, look at all submenus you own (and menu items below that), and if you find one with the specified identifier, send this message to it." This can be used to make communicating with menu items easier, because you don't have to find the handle of every item you wish to send a message to. Using this "include submenus" flag, all you need to do is simply send the message to the Action Bar with the flag set to `TRUE`.

This brings up a design point about menus. It is best to give a unique identifier to each submenu and menu item in a particular application. This identifier serves two purposes. The message parameter of the `WM_COMMAND` contains this value to indicate which menu item was selected. The value is also used to obtain a handle to that item. Using unique IDs ensures that no message intended for a specific control goes astray and that no `WM_COMMAND` is misinterpreted: One identifier, one item.

The way these menu messages are sent is with a call to `WinSendMessage`. What the message is and where you wish it to go determine the method used. For messages with the include submenus flag, the message can simply be sent to the Action Bar, and the submenu search can be responsible for finding the desired item. For the other messages, the handle of the individual item is necessary.

The technique used to obtain the needed handles is similar to obtaining other window handles. The `WinWindowFromID` API works well in these situations.

```
hwndTheMenuBar = WinWindowFromID(hwndFrameMain, FID_MENU);
```

The key in this code is `FID_MENU`. This is the system-defined ID for the Action Bar of a frame window. This code fragment returns the handle of the Action Bar for the frame window with the handle `hwndFrameMain`.

This theory may be applied to any frame window that owns an Action Bar. Once that handle has been established, it can be used to send a message to an item using the include submenus flag. It may also be used to navigate through the menus to get

the handle of the menu item of interest. This process requires that `WinWindowFromID` be called to navigate down through the submenus until it arrives at the handle of the menu item of interest. The message may then be sent directly to it.

The messages used for menu manipulation are

Message	Description
<code>MM_DELETEITEM</code>	This message deletes the specified item from the menu.
<code>MM_ENDMENU</code>	This terminates a menu selection.
<code>MM_INSERTITEM</code>	Using a <code>MENUITEM</code> structure you fill in, this message instructs the menu to insert the item into the menu.
<code>MM_ISITEMVALID</code>	A <code>TRUE</code> return from this message indicates that the specified item is selectable.
<code>MM_ITEMIDFROMPOSITION</code>	This returns the ID of an item, given its index position.
<code>MM_ITEMPOSITIONFROMID</code>	This message returns the index position given an identifier.
<code>MM_QUERYITEM</code>	This copies the information for a menu item into a <code>MENUITEM</code> structure you specify. Note that it does <i>not</i> retrieve the item text.
<code>MM_QUERYITEMATTR</code>	This returns the attributes of a given menu item.
<code>MM_QUERYITEMTEXT</code>	This copies the text of a given menu item into a specified buffer.
<code>MM_QUERYITEMTEXTLENGTH</code>	This returns the length of the text of the given menu item.
<code>MM_QUERYSELECTEDITEMID</code>	If there is a menu item selected, this message returns the ID of the item.
<code>MM_SELECTITEM</code>	This selects the specified menu item.
<code>MM_SETITEM</code>	This message copies the information from a <code>MENUITEM</code> structure into a menu item. Note that this message does <i>not</i> copy the text.

MM_SETITEMATTR	This sets a menu item's attributes.
MM_SETITEMHANDLE	This sets an item's handle.
MM_SETITEMTEXT	This updates the text of a menu item.
MM_STARTMENU-MODE	This message begins a menu selection.

A common function is disabling a menu item so that it is unselectable. The way to accomplish this is with the MM_SETITEMATTR message.

```
WinSendMsg((WinWindowFromID(hwndFrame,FID_MENU)), /* Send it to Action Bar */
            MM_SETITEMATTR,                          /* Msg to set attributes */
            MPFROM2SHORT(MENUID,TRUE),                /* Include submenus=TRUE */
            MPFROM2SHORT(MIA_DISABLED,MIA_DISABLED)); /* Set it MIA_DISABLED */
```

Conversely, an item may be enabled by turning off the MIA_DISABLED attribute.

```
WinSendMsg((WinWindowFromID(hwndFrame,FID_MENU)), /* Send it to Action Bar */
            MM_SETITEMATTR,                          /* Msg to set attributes */
            MPFROM2SHORT(MENUID,TRUE)
            MPFROM2SHORT(MIA_DISABLED,FALSE));        /* Enable the item by */
                                                    /* turning MIA_DISABLED off */
```

In both cases just shown, the message is sent to the Action Bar, and because the submenu flag is TRUE, all submenus are searched until the item with the ID MENUID is found. The message will then be directed at that item and the attributes set accordingly. The result of the first operation is that the menu item is displayed grayed and is unselectable. The second piece of code reenables the item and causes it to be displayed in a normal state.

Summary

Menus are how a user may command an application to take action. Unlike dialogs, they are totally user initiated. Menus do nothing more than send a WM_COMMAND message to the application. It is up to you to do the rest.

Menus are used as a way to notify applications that the user wishes to perform an action. This action may be direct, such as exiting the application, or it may wish to create a dialog to obtain the information needed to complete the requested action. Menus are the first line of communication a user has with an application.

S

SECTION FIVE

GRAPHICS PROGRAMMING INTERFACE (GPI)

The Graphics Program Interface (GPI) opens up the world of graphics to a Presentation Manager programmer. The Presentation Manager Graphics Program Interface has a rich set of functions that enables graphics applications to create and maintain pictures. These functions range from bit-level access to access of more complex picture components. Programs written to exploit the Presentation Manager graphics services will bring this rich world of graphics to users.

Numerical data can be expressed in the form of pie charts, bar graphs, or line graphs. You can display pictures and diagrams using a variety of colors and textures. Graphics and text can appear together on the screen. The graphics can be captioned and annotated in a wide range of text sizes and styles. The subject of graphics is complicated. The OS/2 Presentation Manager GPI makes it simpler to create illustrations than many other systems by providing device independence. However, the pictures must get from the programmer's head into the program and, ultimately, to the output device. The GPI isolates programs from having to know settings for each device in order to draw pictures on it.

Throughout this section, the different layers of translation are discussed, starting with the closest level to the device, the device context.

A *device context* is a program's closest tie to the physical hardware. It contains all the device-dependent information needed to allow a program to write on the device. Through the device context, all high-level drawing instructions get translated to device-level instructions.

The next layer up is the *presentation space*. This space contains the device-independent drawing commands that are issued by programs. When a program draws, it is really drawing into the presentation space. The key to transferring the device-independent commands to the device is the association between the device context and the presentation space.

Device contexts and presentation spaces may be created at any time and need not know anything about each other at the time they are created. As you will see shortly, association at creation time is useful in many situations. This association between the device context and the presentation space is where the translation from the "draw a line from x to y" to "turn on pel 19113, turn on pel 19114 . . . turn on pel 19130" takes place.

The drawing commands going to the presentation space come from the graphics primitives. There are primitives to draw lines, boxes, and arcs in different shapes, sizes, and colors. There are also primitives for characters and other shapes, such as markers. Each of these items may be drawn to the presentation space, which, when it is associated with the device context, displays the desired figures on the output device.

There are tools to manage complex pictures once they are created. You will see how you can use graphics segments to compose pictures and how you can use the different file formats, such as bitmaps and metafiles, to further manipulate and store the pictures you create.

The progression this section follows matches your steps in creating drawings. Each chapter relies on concepts presented in the previous ones all the way through the last chapter, which contains more coding examples than its predecessors. Starting with device contexts and working your way through Chapter 21's discussion of graphics storage and printing, you will see how pictures can be created from nothing more than an idea.

Device Contexts

Presentation Manager programs do not have direct access or physical control of the devices they are using. The Presentation Manager mechanism allows many programs to access a single device, such as the screen, concurrently. Each program may have different requirements for text, graphics, and color. The system manages program access to a device through a device context. A *device context* is a logical description of an output device.

Presentation Manager programs use the device context to get device-specific information from a device driver. You must open a device context before a Presentation Manager device driver can be accessed. This chapter discusses the way in which a program obtains a device context. Later chapters in the section describe the way data is sent to each device, once the device context is opened.

The two API calls provided to open a device context are `WinOpenWindowDC` and `DevOpenDC`. A device context handle is returned when an open request is successful. This handle is then used to request the device context to perform functions such as obtaining the capabilities of the supported device and the supported device names and description.

The different types of device contexts are

- screen
- metafile
- memory

- information
- queued printer/plotter
- direct printer/plotter

The screen, metafile, memory, and printer device contexts are discussed in this chapter. The information device context is used to retrieve information about a printer or a plotter. Opening an information device context is done in the same way as a printer or plotter device context and is not treated separately.

DevOpenDC obtains a device context for memory, information, queued printer/plotter, and direct printer/plotter. The syntax of a DevOpenDC call is

```
hdc = DevOpenDC(hab,                /* Anchor block handle */
                device_context_type, /* Type of DC           */
                "*",                 /* Dev information token */
                4L,                  /* Number of elements   */
                (PDEVOPENDATA)&device_open_data, /* Info Structure       */
                NULL);               /* Compatible open DC   */
```

Certain information is required when you use the DevOpenDC function. Some of this information is provided when you use pointers. These pointers are given in the DEVOPENSTRUC structure. The address of the DEVOPENSTRUC structure is passed as a pointer structure in the fifth parameter to DevOpenDC. The following information is contained in the device_open_data structure:

- pointer to the logical address of the device
- pointer to the name of the device driver
- pointer to a structure that contains device-specific information
- pointer to the name of the queue file data type
- pointer to a string containing a comment that may be displayed if the data is spooled
- pointer to the name of the queue processor
- pointer to an optional string of queue processor parameters
- pointer to an optional string of spooler parameters
- pointer to Network parameters; this is provided if you are using a network

The amount of information passed in the structure depends on the device context being opened. This specific information is explained in the appropriate section on each device.

Screen Device Context

A *screen device context* is a device context that deals with the display screen. You can create a device context either for the entire screen or for a window. The `WinOpenWindowDC` function is used to obtain either type of device context.

The function call in the following example takes the desktop window handle as input and returns a device context handle for the entire screen. Because the desktop is treated as just another window, you can say that a screen device context and a window device context are both really window device contexts.

```
hwndDC = WinOpenWindowDC(HWND_DESKTOP);
```

The following example takes the handle of the client area of a window as input and returns a device context handle for that window.

```
hwndDC = WinOpenWindowDC(hwndClient);
```

A program is allowed to open only one device context per window. When the program terminates or the window is closed, the device context is closed, and the handle is invalidated. The desktop device context is closed when your program terminates. It is not possible to destroy the desktop window.

For all other device context types, the `DevOpenDC` API is used. This function requires additional information, such as the logical address and the device driver name. This optional information includes device-specific information that allows the drivers inside the OS/2 system to create the proper translation mechanism for that device. The descriptions that follow show examples of the minimum information required by `DevOpenDC` for the device context.

Metafile Device Context

One method of saving graphics is to place them in a file. This type of file is called a *metafile* and contains the graphics orders used to re-create a picture. The type of device context for this type of object is called a *metafile device context*. It is specified as `OD_METAFILE` in the call to `DevOpenDC`. The open data structure is passed to `DevOpenDC` as shown here.

```
DEVOPENSTRUC  device_open_data[9]={0L,      /* Logical device addr ptr */
                                     "DISPLAY", /* Device driver name ptr */
                                     0L,        /* Device data ptr          */
                                     0L,        /* Queue file data ptr     */
                                     0L,        /* Spool comment ptr       */
                                     0L,        /* Queue processor name ptr */
                                     0L,        /* Queue proc parm ptr     */
```

```

OL,          /* Spooler parameter ptr */
OL  };      /* Network parm ptr */

```

As with all of the following structures, the data values that are not required by the function may be set to NULL.

Memory Device Context

Whenever you plan to work with bitmaps or memory, you must open a memory device context. The device context is specified as OD_MEMORY in the call to DevOpenDC. When you open a bitmap device context, the compatible device context (last) parameter for the DevOpenDC function is provided. For a screen-compatible bitmap it is given as NULL, as just shown. For a printer bitmap, the printer device context handle is provided. An example of the DevOpenDC call for a printer bitmap is shown here.

```

hdc = DevOpenDC(hab,          /* Anchor block handle */
               device_context_type, /* Type of DC */
               "*",          /* Dev information token */
               4L,          /* Number of elements */
               (PDEVOPENDATA)&device_open_data, /* Info structure */
               hprtDC);      /* Printer DC handle */

```

Regardless of the type of memory device context being opened, the information structure is required. An example of the structure is shown here.

```

DEVOPENSTRUC  device_open_data[9]={0L, /* Logical device addr ptr */
                                   "MEMORY", /* Device driver name ptr */
                                   0L, /* Device data ptr */
                                   0L, /* Queue file data ptr */
                                   0L, /* Spool comment ptr */
                                   0L, /* Queue processor name ptr */
                                   0L, /* Queue proc parm ptr */
                                   0L, /* Spooler parameter ptr */
                                   0L  }; /* Network parm ptr */

```

Printer Device Context

A program can send information directly to the printer or can use a print queue. Direct printing bypasses the OS/2 print spooler. Queued printing sends the data to the print spooler's queue for subsequent printing in the order that jobs are received. Programs that use direct printing issue the DevOpenDC call specifying the device context type as OD_DIRECT. The information passed in the information structure is shown here.

```
DEVOPENSTRUC  device_open_data[9]={ "LPT1", /*Logical device addr ptr */
                                     "IBM4201", /*Device driver name ptr */
                                     OL,        /*Device data ptr */
                                     OL,        /*Queue file data ptr */
                                     OL,        /*Spool comment ptr */
                                     OL,        /*Queue processor name ptr */
                                     OL,        /*Queue proc parm ptr */
                                     OL,        /*Spooler parameter ptr */
                                     OL };      /*Network parm ptr */
```

Programs that print through the spooler issue the DevOpenDC call and specify the device context type as OD_QUEUED. The primary difference between these two methods is the logical address passed in the "device open" data structure. The logical device name, LPT1, is specified for direct printing, whereas the logical device queue name, LPT1Q, is used for queued printing. An example of a queued device context structure is shown here.

```
DEVOPENSTRUC  device_open_data[9]={ "LPT1Q", /* Logical device addr ptr */
                                     "IBM4201", /* Device driver name ptr */
                                     OL,        /* Device data ptr */
                                     OL,        /* Queue file data ptr */
                                     OL,        /* Spool comment ptr */
                                     OL,        /* Queue processor name ptr */
                                     OL,        /* Queue proc parm ptr */
                                     OL,        /* Spooler parameter ptr */
                                     OL };      /* Network parm ptr */
```

Additional information may be required, depending on how the queue and printer were customized by the user with the control panel. If you provide additional information, then the information count parameter should be increased from two to the number of parameters you provide in the device open data structure. The information just shown can be obtained by using the following code sample. This code may be used if you do not know the logical device name and the device driver name.

```
mpflen = 512;

/*Obtain printer device name */
pcnt = WinQueryProfileString(hAB,
                             (PVOID)"PM_SPOOLER",
                             (PVOID)"PRINTER",
                             (PVOID)"",
                             (PVOID)pdevnam,
                             mpflen );

if (pcnt != 0 )
{
    pdevnam[strlen(pdevnam)-1] = 0;
    mpflen = 512;
}
```

```

/* Obtain printer profile string */
pcnt = WinQueryProfileString(hAB,
    (PVOID)"PM_SPOOLER_PRINTER",
    (PVOID)&pdevnam[0],
    (PVOID)"",
    (PVOID)pstring,
    mplen );

if (pcnt != 0 )
{
    ptrchr = strchr(pstring, ';');          /* Increment string */
    ptrchr++;                               /* pointer until you */
    ptrchr = strchr(ptrchr, ';');          /* are looking at the */
    ptrchr++;                               /* logical address */
    *(ptrchr + strcspn(ptrchr, ".,;")) = 0; /* LPT1 or LPT1Q. */
    device_open_data.pszLogAddress = ptrchr; /* Load the address */

    ptrchr = strchr(pstring, ';');          /* Search for the */
    ptrchr++;                               /* printer device */
    *(ptrchr + strcspn(ptrchr, ".,;")) = 0; /* name. */
    device_open_data.pszDriverName = ptrchr; /* Load the address */
}
}

```

This code fragment extracts the information from the OS/2 initialization file, OS2.INI.

Summary

Device contexts are the application's lowest-level communication to the various devices to which output is sent. Any time you send graphics output to a device, a device context must be opened for the device. The device context contains the device-specific (or translator) information to turn the graphic images in the presentation space into pictures on the device. This chapter showed the different types of device contexts and the minimum information needed to use each device context.

This chapter just scratched the surface of device contexts. You will soon see why programmers need and use device contexts for placing graphics on the screen.

Presentation Spaces

A *presentation space* is used to manage and control graphics created by a program. A program does not have to worry about the control of some specific device. Instead, the program sends its output to some generic logical output area—a presentation space. A presentation space is a device-independent description of a picture. The presentation space receives the device independent drawing instructions, and the device translation is accomplished by the device context. The presentation space is created with specific coordinate formats for drawing and, when associated with a device context, the presentation space coordinate units are translated to device units.

A program can have access to multiple presentation spaces at any given time. When your program creates a presentation space, the Presentation Manager system allocates a presentation page and associated control data areas from system memory and returns a presentation space handle. A presentation space that already exists may be requested by a program. In this case, a handle to a previously created presentation space is returned. The presentation page is the internal representation of the output media, and it cannot be directly manipulated by a program. Your programs are only able to reference the entire presentation space by using its handle. All graphics calls that reference a presentation space require the presentation space handle as a parameter. The Presentation Manager system determines whether to access the presentation page or the presentation space data area, and it performs the requested function.

A presentation space must be connected to a device context before any graphics directed to the presentation space appear on the physical output device. Establishing this connection is called “associating” the presentation space with a device context.

Most graphics calls directed to an unassociated presentation space do not return an error, but the program behaves as if the calls were never issued.

The different types of presentation spaces that are available to Presentation Manager programs are discussed in this chapter. We discuss how to allocate or create a presentation space to use in your programs and how to associate a device context with a presentation space.

There are three types of presentation spaces:

- Normal presentation space
- Micro presentation space
- Cached micro presentation space

Normal Presentation Space

Every GPI function may be used with a *normal presentation space*. A normal presentation space is needed if you use graphics segments in your program. Graphics segments are used to manage complex pictures. The primary benefit of graphics segments and normal presentation spaces is that pictures in your programs can be updated at will.

A normal presentation space must always be created by your program. When you create a normal presentation space, the Presentation Manager system allocates the presentation page and control data areas from system memory. The memory allocated by this function for the control data area is created large enough to support graphic segments. More than one normal presentation space may be created by a program. Normal presentation spaces are intended to be used by a program for a long period. They are usually created during program initialization and destroyed during program termination. A normal presentation space can be used to send data to any device type.

In addition, a normal presentation space may be disassociated from a device context and subsequently associated with another device context. The picture is then drawn on the new device or medium.

Micro Presentation Space

A *micro presentation space* is a subset of the normal presentation space. The micro presentation space lacks the support for graphic segments that a normal presentation space provides. Micro presentation spaces must be associated with a device context when they are created and cannot be disassociated, whereas a normal presentation space

may be disassociated from one device and associated with another. A micro presentation space can be used to send data to any device type.

When a micro presentation space is created, the Presentation Manager system allocates a presentation page and control data areas from system memory. The memory allocated for the control data areas is less than that of the normal presentation space because the increased memory required to support graphic segments is not needed. A micro presentation space can be created faster than a normal presentation space, providing a slight gain in application performance.

Micro presentation spaces are designed to be used for short periods. The following represents a typical scenario performed by a graphics routine in your program:

1. Create a micro presentation space.
2. Perform the drawing.
3. Destroy the micro presentation space.

Once the drawing is generated, the presentation space is no longer needed. The picture just drawn does not go away when the presentation space is destroyed, but that region of the window is no longer accessible to the program. Another presentation space must be created either to redraw the picture or to draw something else in that region of the window.

Cached Micro Presentation Space

A *cached micro presentation space* is a micro presentation space kept in a special pool that is maintained by the system. The Presentation Manager system has 16 micro presentation spaces in its pool. These are intended for programs that need a presentation space for a short time—typically no longer than the time it takes to handle a message.

A cached micro presentation space has the same features as a micro presentation space. The difference is that a cached micro presentation space is not created by the program but is maintained and managed by the Presentation Manager system. This provides an even further performance gain because your program doesn't have to create the presentation space before it is used. Because the system maintains these presentation spaces, all a program has to do is ask for one.

A cached micro presentation space is allocated using either the WinGetPS or the WinBeginPaint APIs. The WinBeginPaint API should only be used during the processing of a WM_PAINT message. The WinGetPS may be used anywhere in a program. The following are examples of how these functions are used.

```
hps = WinBeginPaint(hwnd, NULL, rect);
```

or

```
hps = WinGetPS(hwndClient);
```

When one of these functions is called, the system obtains a presentation space from its pool on behalf of the program and associates that space with the device context of the window handle passed on the call to the function. If the handle returned is valid (not NULL), then a presentation space has been successfully allocated, and you are now ready to update the window.

After the window update is complete, the use of the matching half of these functions is required to deallocate the presentation space. The WinBeginPaint API requires the use of the WinEndPaint call, and WinGetPS requires the use of WinReleasePS. The only parameter to these calls is the presentation space handle that is to be deallocated.

```
WinEndPaint(hps);
```

or

```
WinReleasePS(hps);
```

Creating Normal and Micro Presentation Spaces

When a presentation space is created, your program is actually defining the presentation space to be managed by the system. A micro or normal presentation space is defined by specifying the following information:

- the anchor block handle
- the device context handle
- the presentation page size
- the options to be used by the presentation space

The *anchor block handle* is the handle returned when the WinInitialize function is called at the start of the program.

The *device context handle* is required under one of the following conditions:

- A micro presentation space is being created.
- A normal presentation space with a size of zero is being created.
- A device context handle for the device has been obtained and that device context is associated at the time the normal presentation space is created.

If you are creating a normal presentation space, you may specify NULL and associate the presentation space later. The GpiAssociate API is used to associate a presentation space with a device context.

The size of the presentation page is specified in multiples of the coordinate units. The different types of coordinate units are discussed later in this section.

If the size of the presentation space is zero, the presentation space must be associated with a device context at the time it is created. In this case, the size of the presentation space defaults to the maximum size of the output device. For the display, the size defaults to the size of a maximized window. For a printer or plotter, the size defaults to the paper size of the device.

The maximum size of a presentation page depends on the coordinate format selected. You may specify coordinates that are greater than the size of the associated device context. However, only the portion of the presentation page that maps to the device context can be sent to the output media.

For example, suppose the size of the window device context is 640×480 pels, and you specify a presentation page size of 1200×800 . The area beyond 640×480 in the presentation page is not displayed.

Unless you wish to manually manage the area of a picture beyond the size of the device context or to use the additional Presentation Manager transform functions, you should keep the size of the presentation space the same as the device context.

The options that may be specified when you create a presentation space are

- presentation page coordinate units of increment
- coordinate format
- presentation space type
- associate flag

The coordinate units are called *world coordinates*, because they can be applied to real world measurements. Coordinates are used to reference points on the presentation page. The use of the coordinates helps to ensure accurate placement of the graphics image on the presentation page.

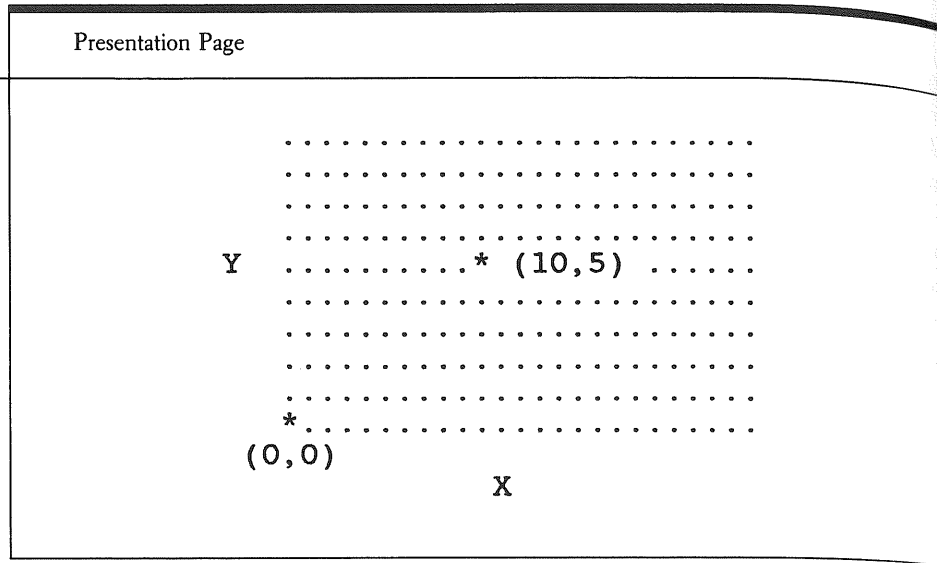
These positional coordinates are given in the form of (x,y) coordinates. The x coordinate references the horizontal and the y coordinate references the vertical axis. The lower left corner of the presentation page is the origin, coordinates (0,0). The x coordinate is always specified first.

Coordinates indicating the origin of the presentation page (0,0) and another coordinate position (10,5) are shown. See Figure 16.1 for a sample of a presentation page.

Three types of coordinate units are available.

- Pels
- Metrics
- Arbitrary

Figure 16.1



Pels

Pels are used when you work with display devices. A pel is the same as a pixel on the display. Not all displays have the same type of pels, so pictures may look different on different displays. A circle, for example, may appear round on a display with square pels, and elliptical on a display that does not have square pels. If these differences are important to your program, the DevQueryCaps function can determine the resolution in pels per meter, and you can use this information to modify your picture.

Metric Units

Metric units are physically measurable units. The units of measurement are

increments of 0.1 millimeter

increments of 0.01 millimeter

increments of 0.01 inch

increments of 0.001 inch

increments of 0.0006944 inch

When you use the metric unit coordinates on a printer or plotter, you can be assured that the drawing will be produced in the specified size. The same cannot be said of display devices, however, as not all pels are the same size. Pels are a relative measurement; the metric units are physical.

Arbitrary Units

Arbitrary units are provided so you can produce drawings with no particular measurement in mind. They enable you to draw pictures as large as possible and preserve the aspect ratio of the drawing on any device. An example of using arbitrary units is a case in which you wish to create a picture using an increment where one unit is equal to the thickness of ten dimes. This is most definitely not a standard size, but using arbitrary units, you may specify the unit to be any size you wish.

The coordinate format is specified in one of two formats: short or long. The short format is two bytes long and has a possible coordinate value range of -32768 through $+32767$. The long format is four bytes long and has a possible coordinate value range of -134217728 through $+134217727$. If no format is specified, the long form is used.

The Presentation Space Type indicator enables you to specify whether a micro presentation space or a normal presentation space should be created.

With the “associate flag” you indicate whether the created presentation space should be implicitly associated with a device context. If you are creating a micro presentation space, you are required to specify a device context handle and specify the associate option by setting the `GPIA_ASSOC` bit.

If a normal presentation space is being created with a size of zero, the associate option must be specified, and the `GPIA_ASSOC` flag must be set. A device context handle must also be provided with the call. If a normal presentation space is to be created at present but associated with a device context later, the “no associate” option is used by setting the `GPIA_NOASSOC` flag. In this case, you should specify `NULL` for the device context handle.

Once you successfully create the presentation space, the system will return a presentation space handle.

A single presentation space can be associated with only one device context at any given time. The function does not work if an attempt is made to associate a presentation space that is currently associated with a device context with a different device context.

There is a way to use a single presentation space with many device contexts. The scenario that may be used to accomplish this function is

1. Associate the presentation space with a device context.
2. Send the output to the presentation space.
3. Disassociate the presentation space from that device context.
4. Associate the presentation space with the next device context.
5. Perform the output to the presentation space.
6. Disassociate the presentation space from the next device context.

A micro or normal presentation space is created by the GpiCreatePS function. The code that can be used to create a micro presentation space is shown in Listing 16.1.

Listing 16.1

```

PSSize.x = 640;
PSSize.y = 480;
hdc = WinOpenWindowDC(hwndClient);          /* Get a device context for window */
hps = GpiCreatePS(hab,                        /* Anchor block handle             */
                  hdc,                        /* Device context handle           */
                  PSSize,                    /* Presentation space size         */
                  PU_PELS ! GPIF_LONG !      /* options are Pels, Long format  */
                  GPIT_MICRO ! GPIA_ASSOC); /* micro PS, and associate        */

```

A normal presentation space associated with a device context may be created as shown in the following listing.

Listing 16.2

```

PSSize.x = 640;
PSSize.y = 480;
hdc = WinOpenWindowDC(hwndClient);          /* Get a device context for window */
hps = GpiCreatePS(hab,                        /* Anchor block handle             */
                  hdc,                        /* Device context handle           */
                  PSSize,                    /* Presentation space size         */
                  PU_PELS ! GPIF_LONG !      /* options are Pels, Long format  */
                  GPIT_NORMAL ! GPIA_ASSOC); /* Normal PS, and associate        */

```

A normal presentation space that will be associated with a device context at some later time may be created as shown in Listing 16.3.

Listing 16.3

```

PSSize.x = 640;
PSSize.y = 480;
hdc = WinOpenWindowDC(hwndClient);          /* Get a device context for window */
hps = GpiCreatePS(hab,                        /* Anchor block handle             */
                  NULL,                      /* Device context handle           */
                  PSSize,                    /* Presentation space size         */
                  PU_PELS ! GPIF_LONG !      /* options are Pels, Long format  */
                  GPIT_NORMAL ! GPIA_NOASSOC); /* Normal PS, and don't associate */

```

Summary

A presentation space receives the device-independent drawing information for graphics output. When the presentation space is associated with a device context, the device-independent information is mapped to the specific hardware. This allows programs to use special features of the available hardware without the need to know the hardware specifics.

A presentation space may be disassociated from a device context and reassociated with another device context to draw the same graphic on different devices. This level of abstraction is useful for taking advantage of the features of each hardware device.

Each type of presentation space is used for a different purpose. Most of the trade-offs involved in your selection are for drawing performance versus processing time. The presentation space can be thought of as your graphics canvas.

Graphics Primitives

Once a presentation space has been created and associated with a device context, you are ready to start constructing pictures. The Presentation Manager Graphics Program Interface provides functions that can draw a variety of shapes, all of which may be combined to form complex pictures. These picture elements are called *graphics primitives*.

Graphics primitives are affected by a set of attributes each possesses and also by the attributes of the presentation space. All attributes have default values that are set should you not specify a specific value. The `GpiQueryXXXX` functions are used to determine the current setting of an attribute while the `GpiSetXXXXX` functions can be used to change the current setting attributes.

Graphics primitives belong to one of six classes. These classes are

- arcs
- areas
- characters
- images
- lines
- markers

Figure

Arc Primitives

Arc primitives enable you to draw circles, ellipses, curves, and wavy lines in your pictures. Each of these shapes can be produced by a single call to the GPI function. Arc primitives are divided into two groups. The first group uses a center reference. The second group uses reference lines or points.

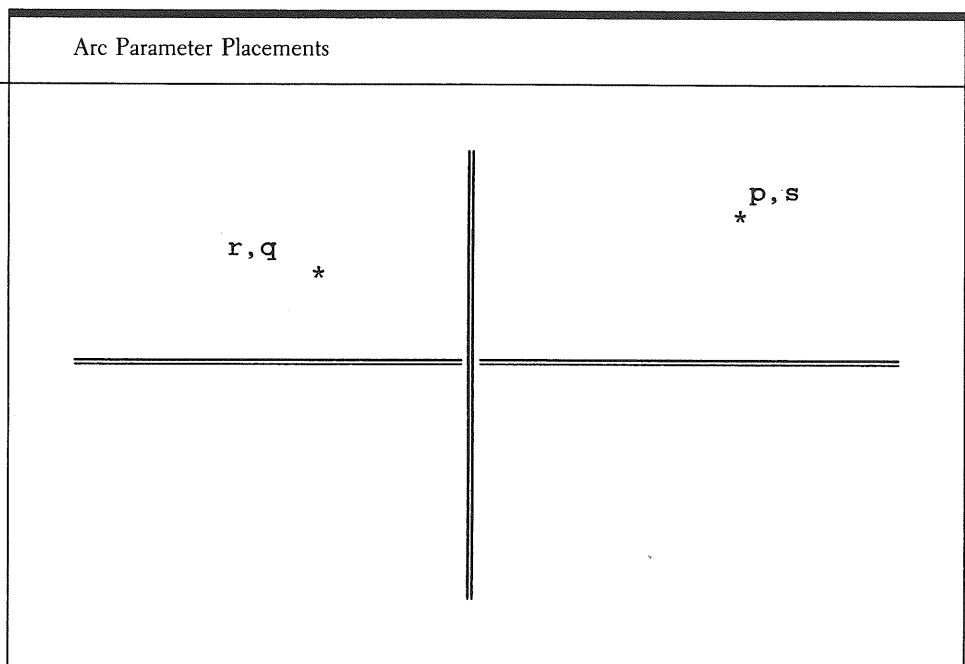
When a circle, an ellipse, or an arc is to be drawn, the current cursor position is used as the center of the figure. The shape of the figure and the direction of drawing are influenced by the current arc parameters.

Arc parameters are specified by four variables p , q , r , and s , as shown in Figure 17.1. The default values are $p = 1$, $s = 0$, $r = 0$, and $q = 1$. These default values may be modified by using the `GpiSetArcParams`. The arc parameters determine the shape and orientation of the arc drawn.

Drawing takes place in a counterclockwise direction when $(p * q)$ is greater than the value $(r * s)$. Drawing is done in a clockwise direction when $(p * q)$ is less than $(r * s)$, and a straight line is drawn when $(p * q)$ equals $(r * s)$.

The values (r,q) and (p,s) are used as coordinates from the origin $(0,0)$ to determine whether a circle or an ellipse is drawn. If the line from $(0,0)$ to (r,q) is the same length as the line from $(0,0)$ to (p,s) , a circle is drawn. Each line is the radius of the circle. If the lines are not equal, an ellipse is drawn.

Figure 17.1



In the case of an ellipse, the line from (r,q) to (0,0) is one half the minor axis of the ellipse, and the line from (p,s) to (0,0) is one half the major axis of the ellipse.

The GpiFullArc API is used to draw a circle or an ellipse. The control parameter determines the appearance and the multiplier parameter determines the size. The figure can be drawn in one of several ways:

- interior filled with no border
- unfilled but with a border
- interior filled with border

The third parameter in the GpiFullArc API is called the *multiplier*. This is a value used to scale the figure being drawn. The multiplier is used to increase the size of the figure beyond that given by the current arc parameters while using the arc aspect of those parameters. With the current arc parameter set to the default values, the following example would create a circle with a filled interior and border, and a radius of 50 units centered at coordinates (100,150).

```
POINTL point;                /* Defining the point variable */
.
point.x = 100;
point.y = 150;
GpiMove(hps,&point);          /* Set current position to (100,150) */
GpiFullArc (hps,DRO_OUTLINEFILL,0x00320000); /* Set scaling factor */
```

The multiplier is a fixed value. To calculate this value, an imaginary binary point is placed between the second and third bytes, and the value 64K for the multiplier equals 1 unit. Integer values are specified as 64K or greater, and fractional values are specified as values between 0 and 64K. So, to scale the figure down by half, you specify 32K in the function call. To double the scale of the figure, you specify 128K. To scale the figure 1.5 times, you specify 96K, and so on.

The circle that is drawn by this code may be changed to an ellipse by modifying the arc parameters to double the horizontal axis. This changes the shape of the figure to elongate it. This code is shown in Listing 17.1.

Listing 17.1

```
ARCPARAMS arc_param;         /* Arc parameter definition */
POINTL point;                /* Defining the point variable */
.
GpiQueryArcParams(hps,&arc_param); /* Get the arc parameters */
arc_param.p = 2;              /* double horizontal coordinate */
                               /* of major axis */
GpiSetArcParams(hps,&arc_param); /* Change the arc parameters */
point.x = 100;
point.y = 150;
```



```
GpiMove(hps,&point);          /* Set current position to (100,150) */
GpiFullArc (hps,DRO_OUTLINEFILL,0x00320000);
```

An arc may also be created using either the `GpiPartialArc` or `GpiPointArc` function call. The `GpiPartialArc` function is used for creating pie-shaped figures. This function begins by drawing a line from the current position to the point that represents the start of the arc. It then draws the arc and sets the current position to the end point of the arc just drawn. The pie shape can be completed by calling `GpiLine` to draw a line from the end of the arc to the initial position.

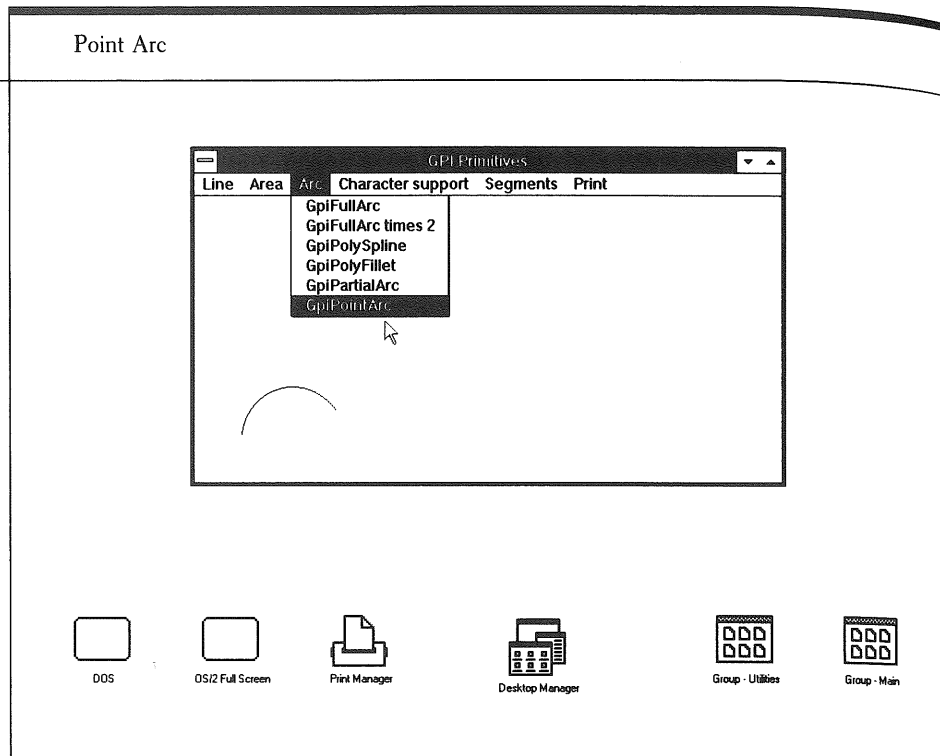
```
POINTL point;                  /* Defining the point variable */
.
.
point.x = 100;
point.y = 50;
GpiMove(hps,&point);           /* Set current position to (100,50) */
point.x = 100;
point.y = 50;
GpiPartialArc(hps,
               &point,         /* Center of arc coordinate */
               0x00500000,      /* Current arc parameter Multiplier */
               0x001E0000,      /* Starting angle of 30 degrees */
               0x00800000,      /* Sweep angle of 128 degrees */
               );
point.x = 100;                 /* Set target point to be the original */
point.y = 50;                  /* starting point */
GpiLine(hps,&point);           /* Draw line to complete pie shape */
```

The `GpiPointArc` call is used to draw a simple arc without any other lines. Figure 17.2 shows a simple point arc.

The current position is used as the starting point, and the other points used to draw the arc are passed to the function. After the arc is drawn, the end point of the arc becomes the current position.

```
POINTL point;                  /* Defining the point variable */
POINTL arc_point[2];           /* Defining the arc point array */
.
.
point.x = 50;
point.y = 50;
GpiMove(hps,&point);           /* Set current position to (100,50) */
arc_point[0].x = 100;
arc_point[0].y = 100;
arc_point[1].x = 150;
arc_point[1].y = 75;
GpiPointArc(hps,
             &arc_point[0]     /* Center and ending points */
             );
```

Figure 17.2



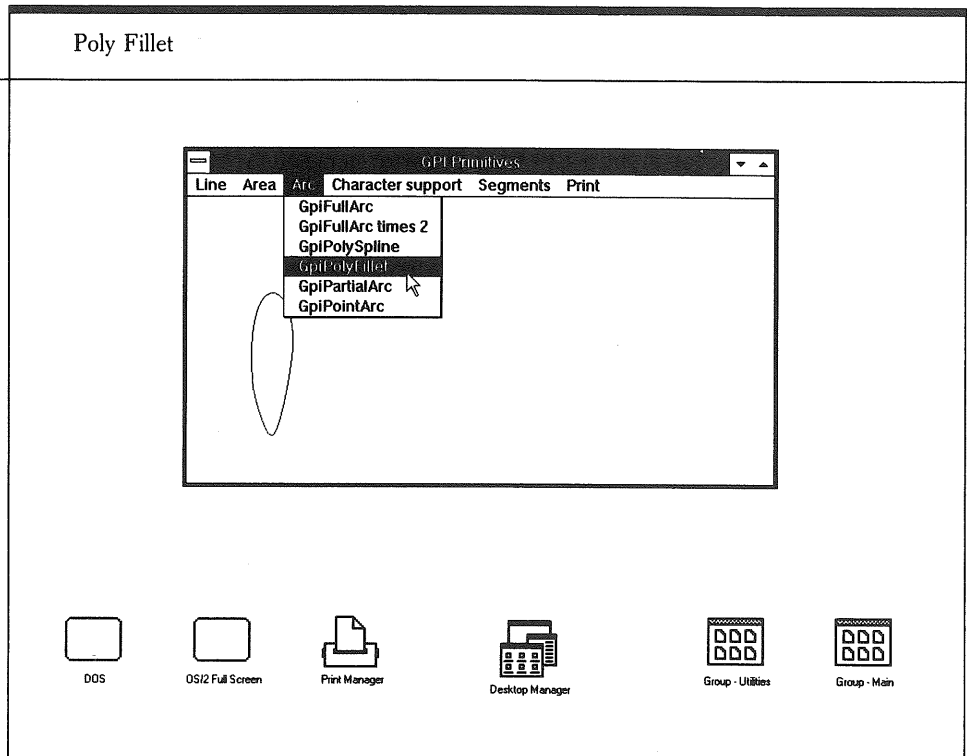
Multiple curved lines or arcs can be drawn using reference lines or multiple reference points. *Reference lines* are imaginary lines constructed from the multiple points passed on the call. The curves are constructed such that they just touch these imaginary lines. The `GpiPolyFillet` function is used to create these types of curved figures.

The “polyfillet” in Figure 17.3 shows how the curves are drawn. It also serves as a comparison for the “polyspline” figure, and it shows how the use of the points differs. The following code fragment shows how to create an egg-shaped diagram with `GpiPolyFillet`.

```
POINTL point;                                /* Defining the point variable */
POINTL arc_point[5]={60,200,                  /* Defining the array of points */
                    120,200,
                    100,50,
                    80,50,
                    70,100};

.
.
point.x = 70;
point.y = 100;
```

Figure 17.3



```

GpiMove(hps,&point);          /* Set current position to (70,100) */
GpiPolyFillet(hps,
              5L,              /* Five points given on this call */
              &arc_point[0]    /* Points used to construct imaginary */
              );               /* lines */

```

A smooth curve can be created by the use of multiple reference points. This type of curved figure is called a *Bezier curve* or *spline*. Each curve is drawn using four points.

The starting point of the curve is the current position; the end of the curve is the third point. The curve is drawn such that the imaginary line between the current position and the first point in the list of four is a tangent to the starting point. The imaginary line between the second point and the third point is tangential at the third point. The curve does not touch point one and point two and the imaginary line between points one and two is not tangential to the peak of the curve. The intermediate points, points one and two, are used as control points to determine the shape of the curve.

Once the curve is drawn, the current position is set to the third point and the next three points in the array are used to draw the next curve. This array of points must

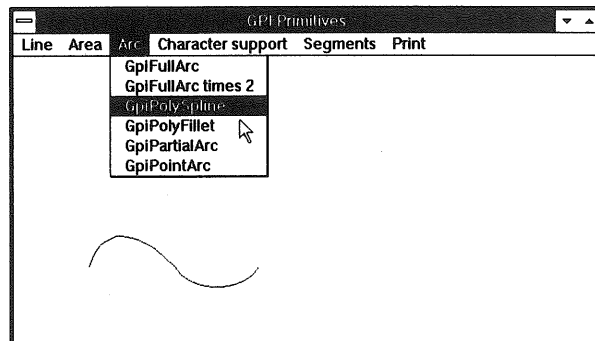
be given in multiples of threes. The GpiPolySpline call is used to draw this type of curve. The final product of this function is shown in Figure 17.4.

```
POINTL point; /* Defining the point variable */
POINTL arc_point[6]={ 10,14, /* Defining the array of points */
                      15,10,
                      17,8,
                      19,2,
                      24,2,
                      26,8};

point.x = 8;
point.y = 8;
GpiMove(hps,&point); /* Set current position to (8,8) */
GpiPolySpline(hps,
              6L, /* Six points given on this call */
              &arc_point[0] /* Points used to control the curves */
              );
```

Figure 17.4

Poly Spline



DOS



OS/2 Full Screen



Print Manager



Desktop Manager



Group - Utilities



Group - Main

You have seen that arcs are used to create pie shapes, circles, ellipses, and just about any kind of curve you may desire. Along with the vast number of curves that can be drawn comes a complex array of points. Although there are no specific attributes for arcs, there are attributes for the other classes of primitives that, when set, affect the appearance of arcs. One such attribute is the line-type attribute, which is discussed shortly.

Area Primitives

One of the benefits of working with graphics is the flexibility to select a variety of colors and patterns. Different colors and patterns are frequently used to present statistical data in a graphical manner.

To use colors and patterns in your applications, boundaries of the shapes must be defined. The boundaries must define a closed figure. The term *area* is used to reference this closed shape figure. Line and arc drawing function calls are used between the GpiBeginArea call and the GpiEndArea call to define the boundaries of your choice. An area that you define can have any size or shape.

The way in which an area is drawn is determined by the options specified in the GpiBeginArea function call. The options are

whether a boundary should be drawn

which mode should be used to construct the interior—alternate or winding

When BA_ALTERNATE is specified, “alternate” mode is used to draw the area. The area is treated similar to a bull’s-eye target. Given a position on the target and moving outward toward the perimeter of the target, the number of lines to get to the final boundary of the area is counted. If the number of a line is odd, that part of the area is filled. If the number of a line is even, no filling is done. Figure 17.5 shows an area with three lines. There the 1’s area is filled in, the 2’s area is not filled, and the 3’s area is filled in.

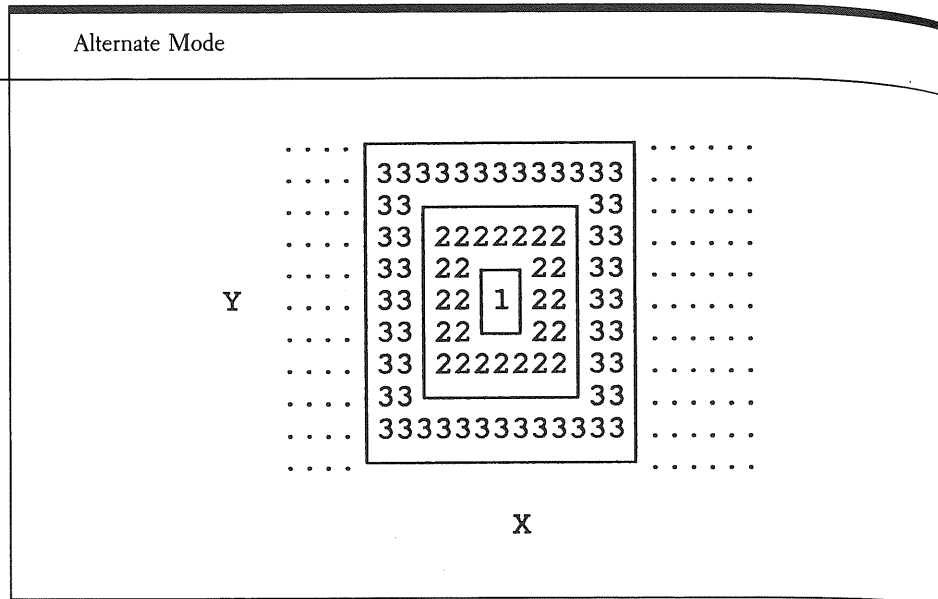
The following is an example of the code necessary to define an alternate mode area.

Listing 17.2

```
LONG   Hround,Vround = 0;          /* Box corner control          */
POINTL point;                      /* Defining the point variable  */

.
.
.
GpiBeginArea(hps,                  /* Start area definition with    */
              BA_NOBOUNDARY ;      /* no boundaries and            */
              BA_ALTERNATE);       /* use alternate mode           */
point.x = 10;
point.y = 4;
GpiMove(hps,&point);               /* Set current position to (10,4) */
```

Figure 17.5



```

point.x = 12;
point.y = 6;
GpiBox(hps,DRO_OUTLINE,          /* Draw the one-box outline with */
        &point,Hround,          /* squared corners and the diagonally */
        Vround);                /* opposite corner at (12,6) */

point.x = 7;
point.y = 2;
GpiMove(hps,&point);             /* Set current position to (7,2) */

point.x = 15;
point.y = 8;
GpiBox(hps,DRO_OUTLINE,          /* Draw the two-box outline with */
        &point,Hround,          /* squared corners and the diagonally */
        Vround);                /* opposite corner at (15,8) */

point.x = 4;
point.y = 0;
GpiMove(hps,&point);             /* Set current position to (4,0) */

point.x = 18;
point.y = 10;
GpiBox(hps,DRO_OUTLINE,          /* Draw the three-box outline with */
        &point,Hround,          /* squared corners and the diagonally */
        Vround);                /* opposite corner at (18,10) */
GpiEndArea(hps);                /* End the definition of the area */

```

Figure

When BA_WINDING is specified, the “winding” mode is used to draw the area. When winding mode is used, an approach similar to that used by alternate mode is

used to determine how the area is filled. The difference between the two modes is that in winding mode, the direction of the imaginary outward lines is taken into account. A counter is incremented when a line is "drawn" in one direction. The counter is decremented when a line is drawn in the opposite direction. If the final counter is zero, that part of the area is not filled; otherwise, it is filled. Figure 17.6 has an area with three lines. In that figure, the 1's area is filled in. The 2's area is filled in only if the line between the 2's and the 3's was drawn in the same direction as the line around the 3's, and the 3's area is filled in.

The following code example re-creates Figure 17.5 using the "winding" mode.

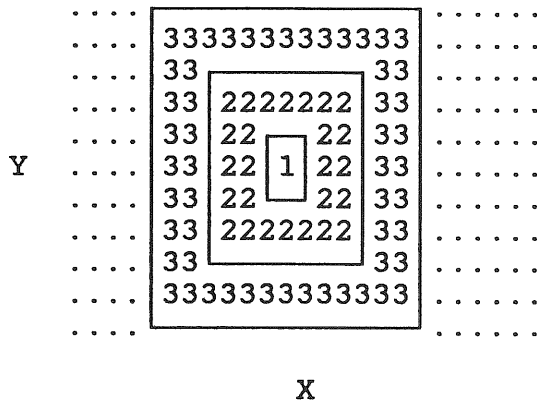
```

LONG   Hround,Vround = 0;           /* Box corner control          */
POINTL point;                       /* Defining the point variable  */
.
.
.
GpiBeginArea(hps,                   /* Start area definition with    */
              BA_NOBOUNDARY ;       /* no boundaries and            */
              BA_WINDING);          /* use winding mode             */
point.x = 10;
point.y = 4;
GpiMove(hps,&point);                /* Set current position to (10,4) */
point.x = 12;
point.y = 6;
GpiBox(hps,DRO_OUTLINE,             /* Draw the one-box outline with */
        &point,Hround,             /* squared corners and the diagonally */

```

Figure 17.6

Winding Mode



```

        Vround);                /* opposite corner at (12,6)      */
point.x = 7;
point.y = 8;
GpiMove(hps,&point);            /* Set current position to (7,8)  */
point.x = 15;
point.y = 2;
GpiBox(hps,DRO_OUTLINE,        /* Draw the two-box outline with   */
        &point,Hround,        /* squared corners and the diagonally */
        Vround);              /* opposite corner at (15,2)      */
point.x = 4;
point.y = 0;
GpiMove(hps,&point);            /* Set current position to (4,0)   */
point.x = 18;
point.y = 10;
GpiBox(hps,DRO_OUTLINE,        /* Draw the three-box outline with */
        &point,Hround,        /* squared corners and the diagonally */
        Vround);              /* opposite corner at (18,10)      */
GpiEndArea(hps);               /* End the definition of the area   */

```

If a figure (or area) is not closed when drawing is completed, the system closes it for you. The complexity of the area is up to you and how you wish to define it. The definition of an area may not take place until the definition of another area is completed.

The area primitive is affected by the attributes that were in effect when the area definition was started. You cannot change the attributes while you are defining an area, but you may certainly set them before you start. Unless the attributes are specifically changed, the area defined uses the system defaults or the last setting.

The area attributes are the

foreground color that determines the color used to draw the foreground of the area

background color that determines the color used to draw the background of the area

foreground mix mode that determines how crossing lines are drawn and the color displayed. For example, when blue letters are drawn over yellow lines, this mode determines whether the crossing points are in blue, yellow, or green

background mix mode that determines what happens when an area is drawn over a previously drawn primitive

pattern set that determines whether the default pattern should be used or some other pattern set that you have selected

symbol that determines which symbol to use, when you are using a font for the pattern set. If the default pattern set is being used, the symbol would be one of the 16 default symbols

The pattern reference point is used to change the location to which the pattern will be applied.

Characters

The character primitive governs the display of text in a graphics environment. The characters used in the display of text belong to a library of characters. This library not only contains the characters but also defines how the characters can be used and how they should be drawn. This library of characters is called a *font*. The OS/2 system font contains one font used by default, along with three additional fonts. The system font is a proportionally spaced font. The three additional fonts are Helvetica, Times Roman, and Courier. Other fonts you have created or purchased may be added to this list of fonts.

Font files may contain one or more fonts. All the fonts in a single file belong to what is called a *font group*. Each font in a group is distinguished from the others by the font face name.

There are two classes of fonts in the system: public fonts and private fonts. Whether a font is public or private depends on the actions of the person who installs it. A font is *public* when it is installed through the OS/2 control panel by the user. It is public because any program running in the system can access the font. Any font that has not been installed by the control panel is *private*. It is private because a program must know the exact directory and file name before the font can be used.

There are two types of fonts: image (or raster) fonts and outline (or vector) fonts. We refer to them as image and outline fonts. The way a font is defined is affected by the character box. This character box is an imaginary rectangle that defines the horizontal and vertical space a character may occupy. An image font character has the pels in each character box set to determine what the character looks like. An outline font character has its borders drawn by lines or arc primitives and is filled in.

The following lists some characteristics a font defines:

- the name of the font
- the size of the characters
- whether the font is an outline font or an image font
- if the space occupied by each character is the same, fixed spacing
- if each character requires a different size space, proportionally spaced
- whether the characters can be mixed with graphics
- whether the font has support for subscripts and superscripts

The selection of a specific font is not required in a Presentation Manager application. If your application does not require any special character effects, the system default font will suffice. If you intend to add a variety of custom character displays to your program, you can change the font used to a font of your choosing.

This feature is useful for creating WYSIWYG (what you see is what you get) applications, such as desktop publishing or any others that provide print previews of how data will look when printed.

Different fonts enable users to display text in a variety of styles and sizes according to how they are implemented by the application. You can mix and match fonts to get just the right effect you want for your application. This is demonstrated by the fact that on the same line you can have characters that are $\frac{1}{2}$ inch high right next to characters that are $\frac{1}{8}$ inch high. You can also display text at any angle or even mix text with graphics.

Character mode settings determine whether an image font is affected by the current character attributes. There are three character mode settings, mode 1 (CM_MODE1), mode 2 (CM_MODE2) and mode 3 (CM_MODE3). An outline font is always affected by the current character attributes. An image font may only be used with modes 1 and 2. The character attributes are the character angle, the character box, and the character shear.

Character Angle

The character *angle* is determined by an imaginary line from coordinate (0,0) to some other coordinate, for example, (10,10). When an outline font is drawn, the character is rotated so the baseline is parallel to this imaginary line.

When you change the angle for an image font, the string that is drawn is rendered with respect to the imaginary line, but the characters themselves are not rotated. Character angles are ignored if the mode is CM_MODE1.

Character Box

The character box, as you have seen, is a way of defining the characters in a font. In addition to defining the type (image or outline), the character box also specifies the size of the standard character.

Character Shear

The character *shear* determines the slanting of a character. Ordinarily, characters are vertical, oriented with the y axis. When you use shearing, however, the character sides are slanted from top right to bottom left or from bottom right to top left, depending on the shear angle specified.

Although outline fonts may be sheared in any mode, shearing affects image fonts only in mode 2. The image font character spacing is changed, but the characters are not sheared.

The following are some considerations that should be taken into account when you use fonts:

- If you are displaying a screen full of text very quickly, such as browsing a document with an editor, then you may be more inclined to use an image font. This is because the image font displays a whole line at a time. The outline font has to draw each character using the line and arc primitives so it takes longer to display a screen of text.
- If you want to display text scaled to the size of a picture, you should use an outline font because of its added flexibility.
- If angled text is desired, an outline font should be used, because characters are not rotated using image fonts.
- If you are going to use shearing, you are restricted to outline fonts. The image fonts do not support shearing.

Images

The Presentation Manager Graphics Program Interface allows your programs to create pictures at the bit level in addition to manipulating primitives. The image API, `GpiImage`, is one of those functions that enable you to manipulate graphics at the bit level. An *image* is a rectangular picture of bits, also known as a *bitmap*.

Each bit represents a pel, or picture element of the screen. A bit set to 1 sets the associated pel using the image foreground color and mix; a bit set to 0 sets the associated pel using the image background color and mix. The result of this is a single color image.

The data supplied to the function determines the pel settings from left to right and top to bottom. For example, if a row of the image is not a multiple of 8, it is padded to complete the byte. The length of the data provided must take this padding into account. Otherwise, the picture is not displayed properly.

The image attribute affects only the color and mix of the primitive. The color attribute is used to set either the foreground color or the background color. The mix attribute is used to determine how the foreground or background bits of the image interact with other foreground or background bits defined in your picture.

Line Primitives

Line primitives all produce drawings from a starting point to an ending point. The starting point is always the current position, and the end point is passed as a parameter to the function. The line, polyline, and box primitives all belong to the line class. Two

APIs provided to establish the current position are the GpiMove call and the GpiSetCurrentPosition call.

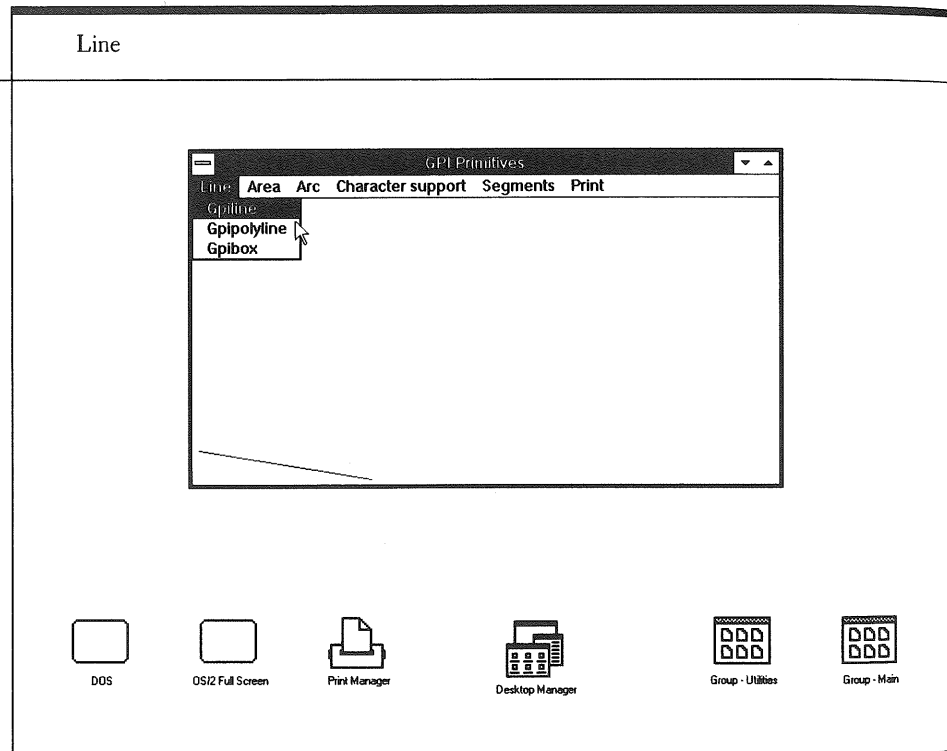
GpiLine is used to draw a single line from the current position to a specified end point. Listing 17.3 shows how to draw the line created in Figure 17.7. In the figure, the current position is (19,3), and the end point is (7,5). After the line is drawn, the end point becomes the current position.

Listing 17.3

```
POINTL point;                                /* Define the point variable */
.
.
point.x = 19;
point.y = 3;
GpiMove(hps,&point);                        /* Set current position to (19,3) */
point.x = 7;
point.y = 5;
GpiLine(hps,&point);                        /* Draw line to end point (7,5) */
```

GpiPolyLine is used to draw multiple connected lines. This can be used in figures such as lines that connect the points of a graph or some other multisided diagram. The

Figure 17.7



Figure

array of points passed to the function provides the points to be used in the figure. As each line is drawn, the end point of the previous line becomes the starting point of the next, until all the points in the array are exhausted. The following code fragment can be used to draw a star shaped figure such as that in Figure 17.8.

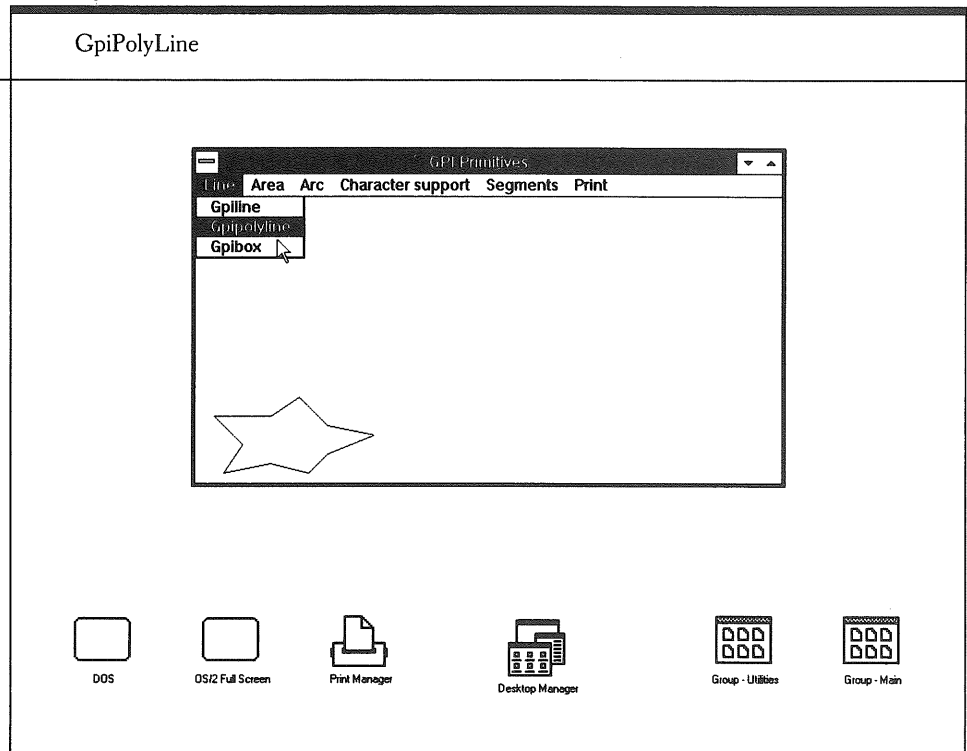
```

/* Define array of coordinates */
POINTL array_point[10]={12,0,
                        14,3,
                        19,5,
                        14,6,
                        11,9,
                        8,7,
                        2,7,
                        5,4,
                        3,1,
                        8,2};

GpiMove(hps,&array_point[9]); /* Set current position to (8,2) */
/* Draw lines to end point (8,2) */
GpiPolyLine(hps,10L,&array_point[0]);

```

Figure 17.8



The last of the line primitives is the box. The `GpiBox` call is used to draw a square or a rectangular shape. The parameters specified on the call determine the type of box that is drawn. You can draw a box that has just a shaded interior and no outline, a box with just the outline drawn, or a combination of the two with a border and shaded interior. It is also possible to specify squared or rounded corners. Listing 17.4 is an example of how to create a box with just an outline and rounded corners.

Listing 17.4

```
POINTL point;                /* Defining the point variable */
LONG   Hround, Vround = 20;   /* Horizontal and Vertical ellipse */
                                   /* axes are both equal to 20 */

.
.
point.x = 9;
point.y = 3;
GpiSetCurrentPosition(hps,&point): /* Set current position to (9,3) */
point.x = 19;
point.y = 7;
GpiBox(hps,DRO_OUTLINE,      /* Draw the box outline with */
       &point,Hround,        /* rounded corners and the diagonally */
       Vround);              /* opposite corner at (19,7) */
```

The current line width setting affects each of the line primitives. The system has a default width that can be changed by the program to any value desired.

The normal or default line width is one pel wide. Only normal line widths are currently supported in OS/2. If your picture requires wide lines, you can achieve the desired results using the `GpiSetGeomLineWidth`, `GpiSetLineEnd`, `GpiSetLineJoin`, and the `GpiBeginPath`/`GpiEndPath` functions. These functions are used to define and draw wide lines.

`GpiSetLineJoin` is used to determine how joined lines appear: beveled, rounded, or mitered. The system default is a beveled joint. The `GpiSetLineEnd` call defines the appearance of the line ends. It is used to specify whether the lines drawn have flat, square, or rounded ends. Square ends are extended by half the width, whereas rounded lines have a rounded end that has a radius of half the width. The default line end is flat. A line three pels wide can be defined and drawn by entering Listing 17.5.

Listing 17.5

```
GpiSetGeomLineWidth(hps,3L); /* Set line width */
GpiBeginPath(hps,1L);        /* Start of path one */
cur_pos.x = 50;
cur_pos.y = 50;
GpiMove(hps,&cur_pos);        /* Set line start position */
cur_pos.x = 150;
GpiLine(hps,&cur_pos);         /* Define line 3 by 100 pels */
GpiEndPath(hps);              /* End wide line definition */
GpiStrokePath(hps,1L,NULL);   /* Draw wide line */
```

Mar

Figure 1

You may substitute the instructions between the `GpiBeginPath` and the `GpiEndPath` calls, using any of the line primitives demonstrated earlier. You may wish to increase the coordinate values previously used to appreciate the effects of the changes.

Markers

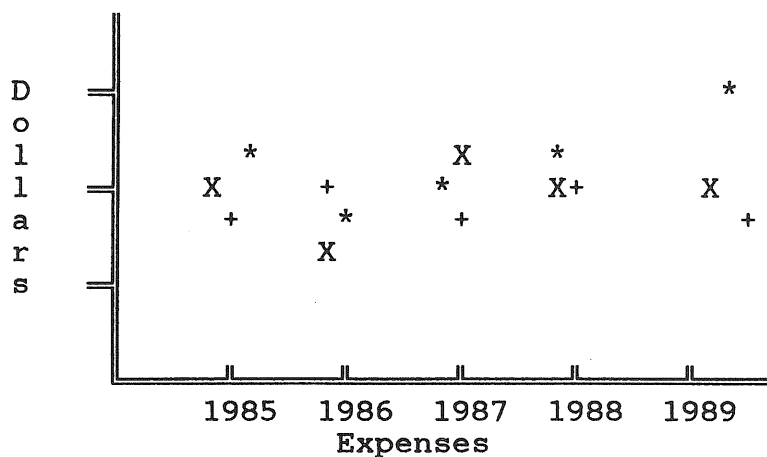
Statistical data can be expressed in many ways. One method that is frequently used to represent this data and give the needed information in a clear and concise manner is a graph. Data items are represented as points on the graph. Each point is represented by a marker. This is demonstrated in Figure 17.9. It shows the expenses for three departments over five years. Marker `X` is used for department one, marker `+` to represent department two, and marker `*` to represent department three.

The `GpiMarker` call is used to draw a single marker centered at the position provided to the function.

```
POINTL point;                      /* Defining the point variable */
.
.
point.x = 5;
point.y = 8;
GpiMarker(hps,&point);             /* Draw a marker at coordinate (5,8) */
```

Figure 17.9

Example of Using Markers



GpiPolyMarker can be used to draw multiple markers with a single function call. Before you issue the GpiPolyMarker call, an array of points that holds the positional coordinate of each point must be created. You are required to also provide a count of how many points are to be drawn.

```
POINTL MK_points[10] = {2,7,      /* Defining the marker points */
                        8,7,
                        11,9,
                        14,6,
                        5,4,
                        14,3,
                        19,5,
                        12,0,
                        3,1,
                        8,2};

GpiPolyMarker(hps,7L,MK_points); /* Draw the first seven markers in */
                                /* the MK_point array */
```

The attributes of the marker primitive are the marker symbol and the marker box. The Presentation Manager system provides 11 marker symbols. GpiSetMarker can be used to change the marker symbol. You are not limited to the system default marker symbol set. A marker can be any character from a font.

The marker box is used to position a marker. The center of the marker box is positioned at the coordinates you specify in both the GpiMarker and GpiPolyMarker calls. The marker box is also the center of the symbol. The size of an image marker cannot be changed, but an outline or vector marker can be scaled by changing the marker box size.

Summary

Graphics primitives are the building blocks of all pictures you may wish to paint. Arcs, lines, boxes, and other shapes may be combined in any way you like to produce graphics.

Each primitive has a set of parameters that define how and where it will be drawn in the windows. OS/2 has defined a set of structures and data types that facilitate the use of these primitives. Using these items we can construct a picture of anything, from a box to a five-story office building.

Graphics Character Support

This chapter discusses different methods used to display text in a window. The three types of APIs available for displaying text are VioWrtXXX calls, the WinDrawText call, and GpiCharXXX calls. The API you choose to use depends on the type of application you are creating and the visual effect you are trying to create. All three APIs can be used in a single window, but the type of presentation space needed varies with which API is used.

The window device context is used with a VIO presentation space to perform VIO writes. The window device context would use a graphics presentation space for WinDrawText or any of the Gpi writing functions.

Displaying Text Strings

Text strings may be displayed in many ways. In DOS and OS/2 full-screen applications, the standard C printf and OS/2 VioWrtTTY functions are all that is needed. In the windowing world of the Presentation Manager interface, the way characters are displayed on the screen can be as simple as using the VioWrtTTY call or can be accomplished with the more powerful Presentation Manager character-drawing functions.

The traditional printf function may not be used to present text in a window. You are now required to use the VIO APIs or the Presentation Manager APIs such as GpiCharStringAt or WinDrawText.

VIO Text Writing

If your program uses a fixed number of rows and columns to display text, it should probably be made an AVIO program. An AVIO program uses a VIO presentation space with an associated window and uses the Vio calls to write text in that window. AVIO programming is discussed in further detail in Section 6.

As a part of a program's initialization, a window is created in which to display text. Once the window has been created, you may open a window device context and create a VIO presentation space to associate with it. A VIO presentation space is needed to write in an AVIO window.

One of the parameters passed to a VIO API is the *video handle*. Programs that do not run in the Presentation Manager session use zero for the VIO handle, whereas AVIO programs use the VIO handle returned from the VioCreatePS function.

Once the VIO presentation space is associated with the window device context, any VioWrt calls that are issued using that VIO handle will cause text to appear in the window. As a part of program termination, the window device context is disassociated from the presentation space by calling VioAssociate and specifying NULL for the device context handle. After the presentation space is disassociated, both the window and the presentation space are destroyed. It is best to do this explicitly in the program, but if this is not done, the system takes care of the cleanup process. An example of this process is shown in this code fragment.

Listing

```

HVPS    VioHandle;                /* VIO handle          */
.
VioCreatePS(&VioHandle,25,80,0,1,0); /* Create a 25 x 80 x 2 PS */
VioAssociate(hwndClientDC,VioHandle); /* Associate VIO PS      */
VioWrtCharStr("Sample VIO string.", /* String to be written  */
              19,                  /* Nineteen bytes to write */
              10,                  /* Presentation Space row 10 */
              15,                  /* Presentation Space col. 15 */
              VioHandle            /* Presentation space handle */
);
VioAssociate(NULL,VioHandle);      /* Disassociate the PS    */
VioDestroyPS(VioHandle);          /* Destroy the PS         */

```

The preceding code assumes that a window has already been created. It starts with the creation of a VIO presentation space that has 25 rows, 80 columns, and one attribute per character. The handle of the VIO presentation space is returned in the variable VioHandle. The VioAssociate function call associates the client area of the window with the VIO presentation space. VioWrtCharStr is then used to display the 19-byte string "Sample VIO string," starting at row 10 and column 15 using the VIO handle. VioAssociate is then used to disassociate the presentation space from the window device context. Then VioDestroyPS is called to destroy the VIO presentation space.

Window Text Drawing

If you want to create a text application using only the windowing API functions and a more refined coordinate resolution, `WinDrawText` is the way to go. The resolution is determined by the presentation page units, such as pels, millimeters, and inches. Resolution is determined when the presentation space is created. Pels are usually used, especially when you work with the display, because they are the smallest addressable display unit.

The initialization of this type of program is similar to the `AVIO` program, except that a graphics presentation space is used instead of an `AVIO` presentation space. `WinDrawText` performs two functions—one to draw text into a window and the other to query the length of a string without actually drawing the text.

Often you will either know the size of the target window or be able to easily determine the window size by using the `WinQueryWindowRect` function. When the window size is known, your program can determine just how much of the string fits on that line by using the query function of the `WinDrawText` call. `WinDrawText` may then be used to draw that portion of the string. An example of this type of function is given in Listing 18.1.

Listing 18.1

```
CHAR achString[512];          /* Character string array.      */
SHORT achStrLen;              /* String length variable      */
SHORT cchWndTxt;              /* Window text character count */
RECT wnd_rect;                /* Rectangle bounding the window */
SHORT indx;                   /* Index variable              */
SHORT cnt;                     /* Count variable              */

.
.
.
achStrLen = sprintf(achString,"This array will be initialized ");
achStrLen += sprintf(achString + achStrLen,"with a very long string ");
achStrLen += sprintf(achString + achStrLen,"of text to demonstrate ");
achStrLen += sprintf(achString + achStrLen,"the use of WinDrawText ");
achStrLen += sprintf(achString + achStrLen,"to format the text in a ");
achStrLen += sprintf(achString + achStrLen,"window. ");
WinQueryWindowRect(hwndClient,&wnd_rect);
wnd_rect.yBottom = wnd_rect.yTop - 15;
/*****
/* Determine how much of the string to the nearest word will
/* fit into the window.
/*
/* The DT_QUERYEXTENT and DT_WORDBREAK flags tell the function*/
/* to see how much of the string will fit and to round it
/* to the nearest word.
*****/
cchWndTxt = WinDrawText(hPS,
                      (SHORT) achStrLen, /* Length of the string      */
                      (PCH) achString,  /* String to be processed    */
                      (PRECTL) &wnd_rect, /* Rectangle used to format text */
```

```

        (LONG)CLR_BLUE,      /* Blue character foreground */
        (LONG) NULL,        /* Default character background */
        (USHORT)DT_QUERYEXTENT | DT_WORDBREAK /* Commands */
    );
    for (cnt = achStrLen / cchWndTxt, indx = 0; indx < cnt; indx++)
    {
        /*****
        /* Draw the first part of the text that will fit into the
        /* window, left justified.
        *****/
        WinDrawText(hPS,
            (SHORT) cchWndTxt, /* Length of the substring */
            (PCH) achString,   /* String to be processed */
            (PRECTL) &wnd_rect, /* Rectangle used to format text */
            (LONG)CLR_BLUE,    /* Blue character foreground */
            (LONG) NULL,       /* Default character background */
            (USHORT)DT_LEFT    /* Left justify text */
        );
        /*****
        /* Shift unprinted part of string to the start and calculate
        /* the next part to be printed.
        /* Set the current position to start on the next
        /* logical line.
        *****/
        achStrLen = sprintf(achString,achString + cchWndTxt);
        WinQueryWindowRect(hwndClient,&wnd_rect);
        wnd_rect.yTop = wnd_rect.yTop - ((15 * indx) + 15); /* New top */
        wnd_rect.yBottom = wnd_rect.yTop - 15; /* New bottom */
        cchWndTxt = WinDrawText(hPS,
            (SHORT) achStrLen, /* Length of the string */
            (PCH) achString,   /* String to be processed */
            (PRECTL) &wnd_rect, /* Rectangle used to format text */
            (LONG)CLR_BLUE,    /* Blue character foreground */
            (LONG) NULL,       /* Default character background */
            (USHORT)DT_QUERYEXTENT | DT_WORDBREAK /* Commands */
        );
    }
    /*****
    /* Draw remainder of string.
    *****/
    if (cchWndTxt <= achStrLen)
    {
        WinDrawText(hPS,
            (SHORT) cchWndTxt, /* Length of the substring */
            (PCH) achString,   /* String to be processed */
            (PRECTL) &wnd_rect, /* Rectangle used to format text */
            (LONG)CLR_BLUE,    /* Blue character foreground */
            (LONG) NULL,       /* Default character background */

```

```

        (USHORT)DT_LEFT    /* Left justify text    */
    );
}

```

This listing uses a standard window and a presentation space that has a presentation page unit of pels. In this sample, a very long text string is used to demonstrate the query functions. An example of where this could be used is to display a buffer of data read from a data file. The size of the window is obtained using `WinQueryWindowRect`. The height of the rectangle to form the box that frames the string about to be written is adjusted according to this returned value. An arbitrary height of 15 pels was used, but you can use the height of the font plus some number of pels as line separators.

The length of the longest string that fits on that line is determined using the `WinDrawText` function with the options `DT_QUERYEXTENT` and `DT_WORD-BREAK`. The length of this string is returned in the variable `cchWndTxt`. A simple algorithm is used to determine how many of these `cchWndTxt` length strings are displayed in the window.

It is determined by dividing the total length of the string, “`achStrLen`,” by the window text length, `cchWndTxt`. `WinDrawText` is used a second time to actually draw the text line in blue and left justified into the window. After each line is drawn, the C run time function `sprintf` shifts the string to the left. A new rectangle is then calculated to display the next string. The top is positioned at the bottom of the previous rectangle, then the bottom is set based on the new top.

Because the string has been shifted to the left, `WinDrawText` is used a third time to determine the new string length that can fit in the window. If any of the string was not processed in the for loop, `WinDrawText` is used one last time to display the remaining string fragment.

`WinDrawText` is like a “one call does it all” for a text application that uses only the Win calls. As demonstrated in this example, `WinDrawText` also justifies the text as it is drawn. The type of justification must be specified as left, centered horizontally or vertically, right, top, or bottom, relative to the bounding rectangle passed on the call. The default justification is left justified.

Graphics Text Functions

Graphics text provides the most flexibility and creative freedom. As you have seen in Chapter 17, graphics text is one class of Presentation Manager graphics primitives. You can control how text is presented. You can control the size, direction, angle, and shear of the displayed text. Other options select a font from those available and control the way the font behaves by selecting the character mode.

Before text can be displayed using the graphics functions, you must determine the effect to create and issue the appropriate Gpi calls to change the character and graphics attributes to produce these results. It is your application’s responsibility to determine

how much space is available to display the text and how long the text to be displayed is.

If you attempt to display a string that would go past the border of a window, the Presentation Manager mechanism clips (truncates) the part of the string that is beyond the window border.

The functions used most often to display characters in a window are the GpiCharXXX calls. These functions enable you to control the text at the string level or at the individual character level.

When you are working at the string level, you can establish the starting position of the string, its angle, and its direction. The system draws the rest of the string based on this information. When you invoke the drawing function, you supply the number of characters to be displayed and the starting address of the first character of the string. The two APIs that provide the string text output functions support are GpiCharString and GpiCharStringAt.

GpiCharString uses the position that was determined before the call was issued. This starting position is the *current position*—the GPI presentation page coordinate where the next primitive starts drawing. When you draw a graphics primitive, the current position is set when the system completes the drawing of that primitive so that the current position is the ending position of the string just drawn. GpiCharString is used primarily when sequential writes to the window are to be done. This is demonstrated in the following code:

```
POINTL  cur_pos;           /* Current position variable */
char    buffer[128]={This buffer is preinitialized.};

.
.
.

GpiSetColor(hPS, CLR_BLUE);    /* Display text in blue    */
cur_pos.x = 10;
cur_pos.y = 10;
GpiMove(hPS,&cur_pos);         /* Set current position    */
GpiCharString(hPS,
              30L,              /* Display 30 bytes        */
              &buffer[0]       /* Start of the string     */
              );
GpiSetColor(hPS, CLR_GREEN);   /* Display text in green   */
GpiCharString(hPS,
              43L,              /* Display 43 bytes        */
              " This will follow the contents of buffer."
              );
```

In this example, the foreground color is set to blue and the current position is set to coordinates (10,10). The text is displayed in blue starting at coordinate (10,10) for 30 characters. After the 30 characters are drawn, the current position is set to the position following the last character. The foreground color is then changed to green, and the

next string is drawn in green. The number of characters drawn is strictly determined by the call's length parameter.

GpiCharStringAt uses the positional parameter of the function call to determine the starting position of the string rather than the current position. This API is most useful when you are randomly writing into the window. This function causes the system to display text one character after the other for the length specified in the length parameter starting at the position specified. The following example shows how you can achieve the same results as the previous example by using the GpiCharStringAt call.

```
POINTL  cur_pos;           /* Current position variable */
char    buffer[128]={This buffer is preinitialized.};

.
.
.
GpiSetColor(hPS, CLR_BLUE); /* Display text in blue      */
cur_pos.x = 10;
cur_pos.y = 10;
GpiCharStringAt(hPS,
                (PPOINTL)&cur_pos, /* Start at position (10,10) */
                30L,              /* Display 30 bytes          */
                &buffer[0]       /* Start of the string       */
                );
GpiSetColor(hPS, CLR_GREEN); /* Display text in green    */
cur_pos.x = 188;
cur_pos.y = 10;
GpiCharStringAt(hPS,
                (PPOINTL)&cur_pos, /* Start at position (188,10) */
                43L,              /* Display 43 bytes          */
                " This will follow the contents of buffer."
                );
```

The difference from the previous example is that it was not necessary to preset the starting position for the first function call. However, the starting position for the second string is required; otherwise, the second string would start at coordinates (10,10) as well, and would overwrite the first string. Neither string would be readable, because when you use the graphics character functions, the previous contents of an area to be drawn are not erased before drawing takes place.

Character-level support enables you to display a string of text while you control the positioning of each character. The APIs that provide this support are the GpiCharStringPos and GpiCharStringPosAt calls. These functions can be used to achieve the same left, center, and right justification results as the WinDrawText call. You are, however, required to calculate the positioning of each character in the string. Except for the first character, the positioning of each character is based on the positioning of the previous character of the string. Each position is given as an offset from the start of the previous character.

GpiCharStringPos is used to start positioning text from the current position. This function does permit different types of positioning: variable, fixed, or mixed.

You use variable positioning when each character's spacing is different. Fixed positioning is best when the spaces between characters are the same, and you employ mixed positioning when you specify both variable and fixed spacing in the same call. For example, if each character is 10 pels wide, GpiCharStringPos can be used to print the string "BarlowT.E.Ricketts" as "Barlow T. E. Ricketts". This is demonstrated by the following:

```
POINTL  cur_pos;                /* Current position variable */
POINTL  apos[80]={10,           /* The "a" in Barlow */
                10,
                10,
                10,
                10,
                30,             /* Position the "T" */
                10,             /* Position the "." */
                20,             /* Position the "E" */
                10,             /* Position the "." */
                40,             /* Position the "R" in Ricketts */
                10,
                10,
                10,
                10,
                10,
                10,             /* Position the "s" in Ricketts */
                };

cur_pos.x = 5;
cur_pos.y = 10;
GpiSetCurrentPosition(hPS,      /* Set current position to (5,10) */
                    (PPOINTL)&cur_pos);
GpiCharStringPos(hPS,
                NULL,
                CHS_VECTOR,
                18L,
                "BarlowT.E.Ricketts",
                (PLONG)&apos[0]
                );
```

As has been demonstrated, the letters in Barlow and Ricketts are positioned in a fixed manner. The spacing between the initials is variable. There are two spaces between w and T., one space between T. and E., and three spaces between E. and R.

GpiCharStringPosAt can start the string at a specific position. GpiCharStringPos and GpiCharStringPosAt display text in a fixed manner even though you may be using

StringPos
be using

StringPos
be using

StringPos
be using

StringPos
be using

[illegible]

```

        10
    };

    GpiSetCharMode(hPS, CM_MODE2); /* Set character mode 2 */
    char_angle.x = 0L;
    char_angle.y = -1L;
    GpiSetCharAngle(hPS, (PGRADIENTL)&char_angle);
    WinQueryWindowRect(hwndW1,&wnd_rect);
    cur_pos.x = 50;
    cur_pos.y = wnd_rect.yTop - 15;
    GpiMove(hPS,(PPOINTL)&cur_pos);
    GpiCharStringPos(hPS,
        NULL,
        CHS_VECTOR,
        24L,
        "Top to Bottom direction.",
        (PLONG)&apos[0]
    );

```

This example assumes that an image font is being used. The characters are not rotated in this example as they would be if an outline font was being used. The example starts by setting the character mode to 2 so that the character box positions the characters. Setting the character angle to $(0, -1)$ causes the current position to change only the y coordinate in a negative direction. The value of the y coordinate is decremented after each character is displayed.

WinQueryWindowRect is used to determine where the top of the window is. The current position is then set to 15 pels below the window top and 50 pels from the left side of the window. GpiCharStringPos is then called to write the string vertically in the window. GpiCharString could have been used, but some of the characters would overlap others and GpiSetCharBox would then have to be used to change the vertical spacing of the characters. The desired result is accomplished by using only one call, GpiCharStringPos.

Changing Fonts

You have decided that you want to use a different font, but do not know how to determine how many fonts there are or how to select a font. Before starting the search for a suitable font, you must be aware that the Presentation Manager system does not simulate a font. The font you wish to use must physically exist before you are able to use it. For example, all the Times Roman and Helvetica fonts are proportionally spaced. You could not select a fixed-space Times Roman or Helvetica font, because it does not exist.

The first step in the process of selecting a new font is to load the private fonts that your program knows about. The three font groups provided by OS/2 are Times Roman, Helvetica, and Courier. If there are others that are going to be provided with your application, they should also be loaded. Private fonts should only be loaded once, at the time the application is initialized. The GpiLoadFonts API is used to load fonts into an application. The following example shows how this can be done.

```

/* Load the Helvetica font */
if (!(GpiLoadFonts(hab,
    "C:\\OS2\\DLL\\HELV.FON")))
    Handle_Font_Not_Present(); /* Report error to user */
/* Load the Times Roman font */
if (!(GpiLoadFonts(hab,
    "C:\\OS2\\DLL\\TIMES.FON")))
    Handle_Font_Not_Present(); /* Report error to user */
/* Load the Courier font */
if (!(GpiLoadFonts(hab,
    "C:\\OS2\\DLL\\COURIER.FON")))
    Handle_Font_Not_Present(); /* Report error to user */
/* Error recovery */

```

Once this code is executed, these three font groups are available to your program.

The next step in the process is to decide which type of font you are interested in, public or private. If you are interested in both, you would call GpiQueryFonts as shown in the following code to determine the number of fonts that are available.

```

LONG FontCount; /* Define the font count variable */

FontCount = GpiQueryFonts(hPS,
    QF_PUBLIC ! /* Select public and */
    QF_PRIVATE, /* private fonts */
    NULL, /* Query all available fonts */
    NULL, /* We don't want metrics */
    NULL); /* information so these=NULL */

```

If you are working with private fonts only, QF_PUBLIC is not needed. Likewise, if only public fonts are required, only QF_PRIVATE is essential. Once the number of available fonts is determined, DosAllocSeg allocates the memory required to hold the font information for the application.

```

PFONTMETRIC fontmet; /* Fontmetric pointer */

if (FontCount != NULL)
{
    /* Allocate font count times the */

```

```

/* font metrics structure size */
DosAllocSeg((FontCount * sizeof(FONTMETRICS)),
            &(SELECTOROF(fontmet)),
            SEG_NONSHARED);
}

```

Once the memory has been allocated, you are now ready to get the information about the available fonts. If you specify the face name, you get information for only that font group. By not specifying the face name parameter, information for all the fonts will be retrieved by GpiQueryFonts. GpiQueryFonts is then called again, but this time with the parameters necessary to get the information for a particular as shown.

```

LONG ReqFont;          /* Define the requested font */
                        /* count variable */
                        */
.
.
if (fontmet != NULL)
{
    ReqFont = FontCount;
    FontCount = GpiQueryFonts(hPS,
        QF_PUBLIC !      /* Select public and */
        QF_PRIVATE,      /* private fonts */
        NULL,            /* Query all available fonts */
        ReqFont,
        sizeof(FONTMETRICS), /* Size of each metric */
        fontmet           /* Requested information */
    );

    /* The search logic follows. */
}

```

Now that the font information is available, it is necessary to look for the characteristics you want. The following sample looks for a font that has the following basic characteristics:

An image font

The character size is 6 pels wide by 9 pels high

The first font that satisfies these requirements is used. The necessary information is saved for future reference.

Listing 18.2

```

FATTRS fontsel;          /* Font selection structure */
.
.
                        /* Font selection logic */
for (indx = 0; indx < ReqFont; indx++)
{
    /* Check for an image font */
}

```

```

if (!(fontmet[indx].fsdefn & 0x8000))
{
    /* Check for a 6 x 9 character */
    if (fontmet[indx].lAveCharWidth == 6L &&
        fontmet[indx].lMaxBaselineExt == 9L)
    {
        /* Set FATTR size */
        fontsel.usRecordLength = sizeof(FATTRS);
        /* Save the match number */
        fontsel.lMatch = fontmet[indx].lMatch;
        /* Save the face name */
        sprintf(fontsel.szFacename, fontmet[indx].szFacename);
        /* Save codepage */
        fontsel.usCodepage = fontmet[indx].usCodepage;
        indx = ReqFont; /* End the search */
    }
}

```

If a font that satisfies the requirements is found, it is selected by this code fragment. The font is selected by creating a logical font and setting it as the current font. The font attribute structure is used as input to the GpiCreateLogFont API to create a logical font. This font attribute structure contains the following information:

- The length of the structure must always be provided.
- The selection indicators indicate whether the characters should be displayed using italic, underscore, bold, or strikeout. One or more of these indicators can be set at once.
- The match number used to make a specific selection is a unique number that is obtained from the metrics of the loaded font.
- The face name of the font is the name given to a certain group of fonts. It is a nonmechanical way of describing certain common style among the fonts of the same group. For example, the face name Courier encompasses all the Courier fonts, but a face name Courier Bold would help you to differentiate between a nonemphasized, light-printing Courier font and an emphasized, dark-printing Courier font.
- The font registry is not used and is always passed as zero.
- The code page is used to determine which language the font supports. A code page is a table used by the system that defines the appearance of a given ASCII value. For example, an ASCII value of B5 in code page 437 looks like Æ but in code page 850 it is Á. Code page 850 is the multilingual table that is used by OS/2.

- The maximum baseline extension determines the maximum height of the tallest character.
- The average character width determines the average width of the character.
- The type indicators are two values: kerning and fixed. The kerning indicator (`FATTR_TYPE_KERNING`) indicates whether some of the characters hang over others, such as Tr. The fonts provided with OS/2 do not provide kerning; however, other externally available fonts (or ones that you may design) can support kerning. The fixed indicator (`FATTR_TYPE_FIXED`) indicates whether a proportional spaced font is required. Setting the fixed indicator means that either a fixed or a proportionally spaced font can be used. If this indicator is not set, then a proportional font is requested. Only the Courier fonts are fixed. All others are proportional.
- The font use indicators tell how the font is intended to be used. The indicators are nomix, outline, and transformable. The nomix indicator, (`FATTR_FONT_USE_NOMIX`), tells the system that the characters are not mixed with graphics. The outline indicator, (`FATTR_FONTUSE_OUTLINE`), tells the system that an outline font is requested. The transformable indicator, (`FATTR_FONTUSE_TRANSFORMABLE`), tells the system that the characters may be scaled, rotated, and sheared.

The minimum required entries are the structure length, the font face name, the code page, and the match number. Once the values in the font attribute structure are initialized, a logical font may be created. The `GpiCreateLogFont` API requires the handle of the presentation space that will be used, the address of the logical font name, the local identification number, and the address of the font attribute structure.

The logical font name can be any eight characters. If you are going to create files that other programs use, and the logical font name is a part of the information that the other program uses to accurately reconstruct your displayed data, then the logical font name should be the file name without the extension of the font you are using.

You provide a local identifier, which is a numeric value between 1 and 254. This ID should be one that is not already being used by your program. If your request to create a logical font is successful, the system returns `FONT_MATCH`. The local identifier is now valid for that presentation space. This ID may now be used as a parameter to the `GpiSetCharSet` call to make this font the current, active font used with that presentation space. If the system cannot find a match, the value `FONT_DEFAULT` is returned. If this occurs, the ID is not valid, and the system default font is used. If you passed invalid parameters or structure information, the call fails, and the system returns a code value of `GPI_ERROR`. If the call fails, you use the `WinGetLastError` call to determine the cause of the failure.

```

lcid = 20; /* Initialize local ID */
rc = GpiCreateLogFont(hPS,
                      "MyLogFnt", /* Logical font name */
                      lcid, /* Local ID for this font */
                      &fontsel); /* Font attribute structure */
if (rc == FONT_MATCH)
    GpiSetCharSet(hPS,lcid); /* Establish current font */
else
    if (rc == GPI_ERROR)
        Process_error(); /* Font was not selected do */
                          /* what you want otherwise */

```

When GpiCreateLogFont is called, the system performs the following verification:

1. The structure length is checked to verify that it is equal to the size of the font attribute structure.
2. The selection, type, and font use fields are checked to make sure you did not specify an invalid flag.
3. If you specified the match number, it verifies the face name and the codepage to make sure that you specified the correct match number.

When you specify a match number, the system does not use any other information from the font attribute structure. There is another method of creating a logical font, but it does not offer as high a degree of certainty and requires that you know a little more about the fonts on your system. The primary difference from the previously discussed method is that you do not have to issue the GpiQueryFonts calls, so you do not have to allocate memory. The minimum font attribute structure initialization required then would be

- The font attribute structure size must be specified.
- The match number would be set to zero.
- The face name must be specified and must be correct. The face names for the three private fonts provided with OS/2 are Helv for Helvetica, Tms Rmn for Times Roman, and Courier for Courier.
- The code page is set to 850, otherwise it must match the font.
- The average character width must be specified.
- The maximum baseline extension must be specified.

If the face name, code page, character width, and baseline extension all match a specific font, the GpiCreateLogFont call will be successful, and the system will return

FONT_MATCH. If they do not match, the system returns FONT_DEFAULT and uses the default font.

The checking of the baseline extension depends on finding a valid character width. A successful font match depends on finding a valid baseline extension. If you use this method and you are trying to create an image font and a suitable match cannot be found, the system tries the outline fonts for a possible match. After a logical font has successfully been created, you can then set it as the current font using the GpiSetCharSet function.

When you have loaded a font, it uses system memory and other Presentation Manager resources. Before you terminate your program, you are required to unload the fonts you have loaded so these resources can be released. They are not released automatically when your program terminates. This is performed by the GpiUnloadFonts function.

```

/* Unload the Helvetica font */
GpiUnloadFonts(hab, "C:\\OS2\\DLL\\HELV.FON");
/* Unload the Times Roman font */
GpiUnloadFonts(hab, "C:\\OS2\\DLL\\TIMES.FON");
/* Unload the Courier font */
GpiUnloadFonts(hab, "C:\\OS2\\DLL\\COURIER.FON");

```

Creating Markers

The Presentation Manager system provides a set of 11 marker symbols. Using the Presentation Manager Gpi calls, you can create custom marker symbol sets. These markers are character symbols that can be used to indicate statistical data points on a graph. If a font other than the default font has been selected, any characters within the font can be used as a marker. This is done by using the same local identification that was used to select the font to determine the marker set. GpiSetMarkerSet is used to set the current font as the current set of marker symbols.

```
GpiSetMarkerSet(hPS, lcid);
```

Once the marker set has been established, the character that is used as the current marker can be selected using the GpiSetMarker API. The ASCII value of the character passed as the second parameter to GpiSetMarker sets that symbol as the current marker. For example, suppose you wanted to use the heart (ASCII character 03) for a symbol. The following code selects 03 as the marker symbol:

```
GpiSetMarker(hPS, 03L);
```

Once this is accomplished, the heart symbol is drawn as the marker at the point specified whenever GpiMarker or GpiPolyMarker is called.

If you want to use some other font besides the current font for the marker set, the same procedure for changing the font is used. The procedure is the same up until the `GpiCreateLogFont` call. Instead of calling `GpiSetCharSet`, `GpiSetMarkerSet` is called to change the marker set, followed by `GpiSetMarker`. Now it is possible to use one font to display text and another font for the marker symbol set.

Creating Patterns

When you work with shaded figures, it is often desirable to change the pattern used to fill figures. The Presentation Manager GPI provides a set of default patterns you can use. If those do not fulfill your requirements, you can change them and use your own. Patterns are used to fill circles, boxes, and areas. The patterns supported by OS/2 1.1 are all 8 pels wide by 8 pels high.

The patterns can either be bitmaps or characters from a font. If a bitmap larger than 8×8 pels is used, or a font with characters larger than 8×8 , the system only uses the first 8×8 portion of the pattern. Using both fonts and bitmaps as patterns is shown in the following examples.

Bitmap Patterns

If you decide to use a bitmap for a pattern, the first step is to create the bitmap that you will be using in your program. Bitmaps can be created in a variety of ways, the simplest of which is the Icon Editor provided in the OS/2 Programmer's Toolkit.

Once the bitmap has been created, it must be made available to your program. To make it available, you add a line to the resource file for the application and assign a resource ID to it.

Before a bitmap can be used as a pattern, it must be loaded. This can be done at program initialization or at the time the function that uses the bitmap is called. However, you only need to load the bitmap once. `GpiLoadBitmap` is used to load a bitmap into memory.

```

HBITMAP hbitmap;           /* Bitmap handle          */
.
.
hbitmap = GpiLoadBitmap(
    hPS,                    /* Bit map PS handle      */
    (USHORT) 0,              /* No Dynalink Module used */
    (USHORT) ID_BMAP,        /* bit map #? will be loaded */
    (LONG) 8,                /* width is 8 pels wide   */
    (LONG) 8                 /* height is 8 pels       */
);

```

Once the bitmap is successfully loaded, a handle to the bitmap is returned to the program. If more than one bitmap is being used, each has a different ID. The bitmap

ID referenced in the previous code, (ID_BMAP), is the identifier that was assigned to the resource in the .RC file. Just like any resource or any window, each bitmap has its own unique handle. The GpiSetBitmapId call is used to establish a local identification for a bitmap, setting it as a pattern is

```
ULONG lcid;                                /* Local identification variable */
.
.
lcid = 40;
if(GpiSetBitmapId(hPS,hbmap,lcid)) /* Assign an LCID to the bitmap */
    GpiSetPatternSet(hPS,lcid);    /* Make the bitmap the current pattern */
```

After you use GpiSetBitmapId, GpiSetPatternSet must be executed to establish the bitmap as the current pattern symbol. Assume that another bitmap was also loaded, and its bitmap handle was hbmap2. If the first bitmap has been used and you wish to use the second bitmap with the same local identification, you can execute the following code to make that change.

```
GpiSetPatternSet(hsPS,LCID_DEFAULT); /* Make pattern 1 current */
GpiDeleteSetId(hsPS,lcid);           /* Remove the connection to lcid 40 */
if(GpiSetBitmapId(hPS,hbmap2,lcid)) /* Connect new pattern to lcid 40 */
    GpiSetPatternSet(hPS,lcid);      /* Set the new pattern current */
```

To make the change, the default pattern set is first made current. The GpiDeleteSetId is called to break the association of the first bitmap with lcid 40. The GpiSetBitmapId links the second bitmap with lcid 40. The second bitmap is set as the current pattern symbol with GpiSetPatternSet.

This is just one of the many ways multiple bitmaps can be used as pattern symbols. You can also use multiple local identifiers. The choice of which method you use is yours and is a part of your application design decision.

Font Patterns

Use of a font for a pattern requires a slightly different approach and is a little more involved. The benefit gained from using a font as a pattern is that there are 255 symbols available without having to change the lcid or pattern set. You can create your own set of customized patterns by creating a pattern font.

To load the pattern font, follow the same procedures discussed earlier in this chapter for changing fonts. First, the font has to be loaded. It is then necessary to query the font metrics, initialize the font attribute structure, and create a logical font. After the logical font is created, GpiSetPatternSet is called to establish the font as the current pattern symbol set. You are then free to set any of the font characters as the current pattern symbol by passing that ASCII character value as the second parameter on the GpiSetPattern call.

Summary

In addition to pictures, the OS/2 Presentation Manager system provides extensive support for drawing characters in windows. There are APIs to control size, string justification, and text positioning. There are also functions to change the fonts that are used to draw the characters. These functions provide a number of ways to display text in the windows to complement graphics.

Graphics Segments

This chapter discusses in detail the use of graphics segments in your program. Before you begin the discussion of segments, the chapter covers the three drawing modes and how they affect your use of segments. The three drawing modes are draw mode, retain mode, and draw and retain mode. Afterward, this chapter covers how to create, edit, and copy graphics segments, as well as the segment attributes and their effect on the segments.

Modes

The three drawing modes dictate the way the primitives will behave and whether you can use some primitives. Some primitives are restricted in certain modes. The drawing mode is actually a presentation space attribute and functions on a per-presentation-space basis. For any normal presentation space only one mode can be in effect at any given time. The mode is established by the `GpiSetDrawingMode` API.

Draw Mode

The default mode is draw. Draw mode can be established explicitly by setting the `GpiSetDrawingMode` mode option to `DM_DRAW`. When a presentation space is created and associated with a device context, the drawing mode established is draw mode. This is the mode that you have seen so far.

This mode is where all primitive functions you use produce an immediate result. For example, when `GpiCharString` or `GpiLine` is called, the string of text or the line drawn is immediately visible on the output device. As a result, it is the fastest of the three modes.

This mode should be used if the type of application you are designing does not have complex graphics that take a long time to create. Only nonretained graphics are allowed in draw mode.

This means that once the primitives and attributes are applied they are lost and cannot be reapplied, unless all the individual primitive and attribute functions are reissued. For example, if you draw a box with 15 lines and then at some point need to redraw it, the box must be redrawn from scratch. Once the primitives are drawn in draw mode, they are gone and must be completely re-created.

Retain Mode

If you are creating an application that makes use of complex pictures that take a long time to create or you are working with graphics constructed in a modular fashion, you should use retained graphics.

Engineering and architectural applications are most suitable for this type of graphics, because they frequently use the same components throughout a picture. The drawing mode for retained graphics is `DM_RETAIN`. To set this mode, use `GpiSetDrawingMode` and specify the mode option as `DM_RETAIN`. Unlike draw mode, the presentation space does not have to be associated when you use retain mode. The drawing instructions are placed in a segment. When the presentation space is later associated with a device context, the segments are then drawn. Retain mode is not as fast as draw mode, but it offers other benefits. The benefits achieved using retained graphics include easy modification of stored graphics, the ability to determine what needs to be redrawn, the ability to redraw just parts requiring regeneration, and the ability to break a picture down into manageable components.

Draw and Retain Mode

The draw and retain mode may offer some performance compromise, because the picture is displayed as it is being created while at the same time being saved for later use. For example, if you draw a box, the box is first drawn on the screen. Then it is added to the segment being created. You can later redraw the box by drawing the segment. Draw and retain mode is set by specifying the `DM_DRAWANDRETAIN` option in a call to `GpiSetDrawingMode`. Draw and retain mode offers capabilities similar to those of retain mode, except that the presentation space must be associated with a device context, and you cannot edit a stored picture.

Creating a Segment

When you work in either retain or draw and retain mode, graphics *segments* are used to store pictures. Retained graphics segments serve the same purpose for your application as a disk file does for data.

Segments give applications the ability to redisplay previously created pictures using a single API call such as `GpiDrawSegment`. A graphics segment is like a program file that contains the instructions that make up a subroutine in an application. The segment contains the primitives and attributes that make up a component of a total picture. For example, an interior design type application may have one segment that draws a chair, another that draws a bed, another that draws a cabinet, and an additional segment that draws a table. Each by itself does not make a room full of furniture, but drawn in conjunction they make up a complete room.

As a part of an application's initialization, segments that will be used should be created. This relieves the overhead of creating a segment when the user selects an application display request.

Before a segment is created, you must establish the drawing mode. This is accomplished using the `GpiSetDrawingMode` API as previously discussed. The attributes set at this time apply to the whole segment.

Once this initialization is complete, the segment can be created by calling `GpiOpenSegment`. The primitives and attributes that are part of the picture in that segment are placed into it; then it is closed with the `GpiCloseSegment` call. Once the segment is closed, it can now be drawn to create the picture (or the part of a picture) the segment is responsible for reproducing. Listing 19.1 demonstrates the process.

Listing 19.1

```
GpiSetDrawingMode(hps,DM_RETAIN); /* Set the drawing mode to retain */
GpiSetBackColor(hps,CLR_BLUE);   /* Set segment background to blue */
GpiOpenSegment(hps,1L);           /* Open segment # 1 */
cur_pos.x = 180;
cur_pos.y = 300;
GpiMove(hps, &cur_pos);           /* Set current position (200,300) */
GpiSetColor(hps,CLR_BROWN);
arc_point[0].x = 200;
arc_point[0].y = 310;
arc_point[1].x = 220;
arc_point[1].y = 300;
GpiPolyFilllet(hps,                /* Draw top of lamp */
                2L,
                &arc_point[0]
                );
/*****
/* Lamp shade definition.
*****/
GpiSetColor(hps,CLR_WHITE);       /* Set shade color to white */
```

```

GpiBeginArea(hps,
              BA_ALTERNATE | BA_BOUNDARY
              );
cur_pos.x = 140;
cur_pos.y = 300;
GpiMove(hps, &cur_pos);
cur_pos.x = 260;
GpiLine(hps,&cur_pos);          /* Draw top of shade          */
cur_pos.x = 320;
cur_pos.y = 230;
GpiLine(hps,&cur_pos);          /* Draw right side of shade   */
cur_pos.x = 80;
GpiLine(hps,&cur_pos);          /* Draw base of shade        */
cur_pos.x = 140;
cur_pos.y = 300;
GpiLine(hps,&cur_pos);          /* Draw left side of shade   */
GpiEndArea(hps);
/*****
/* Lamp base definition.
*****/
GpiSetColor(hps,CLR_BROWN);    /* Set lamp color to brown    */
GpiBeginArea(hps,
              BA_ALTERNATE | BA_BOUNDARY
              );
cur_pos.x = 220;
cur_pos.y = 229;
GpiMove(hps, &cur_pos);
cur_pos.x = 180;
GpiLine(hps, &cur_pos);
arc_point[0].x = 190;
arc_point[0].y = 225;
arc_point[1].x = 190;
arc_point[1].y = 215;
arc_point[2].x = 180;
arc_point[2].y = 210;
arc_point[3].x = 160;
arc_point[3].y = 205;
arc_point[4].x = 160;
arc_point[4].y = 195;
arc_point[5].x = 180;
arc_point[5].y = 190;
arc_point[6].x = 190;
arc_point[6].y = 175;
arc_point[7].x = 180;
arc_point[7].y = 160;
arc_point[8].x = 140;
arc_point[8].y = 155;
arc_point[9].x = 140;

```

```

arc_point[9].y = 135;
arc_point[10].x = 180;
arc_point[10].y = 130;
arc_point[11].x = 180;
arc_point[11].y = 115;
arc_point[12].x = 160;
arc_point[12].y = 100;
arc_point[13].x = 120;
arc_point[13].y = 100;
arc_point[14].x = 100;
arc_point[14].y = 80;
GpiPolyFilllet(hps,                                /* Draw left side of base */
               15L,
               &arc_point[0]
               );
cur_pos.x = 300;
cur_pos.y = 80;
GpiLine(hps, &cur_pos);                          /* Draw baseline of base */
arc_point[0].x = 280;
arc_point[0].y = 100;
arc_point[1].x = 240;
arc_point[1].y = 100;
arc_point[2].x = 220;
arc_point[2].y = 115;
arc_point[3].x = 220;
arc_point[3].y = 130;
arc_point[4].x = 260;
arc_point[4].y = 135;
arc_point[5].x = 260;
arc_point[5].y = 155;
arc_point[6].x = 220;
arc_point[6].y = 160;
arc_point[7].x = 210;
arc_point[7].y = 175;
arc_point[8].x = 220;
arc_point[8].y = 190;
arc_point[9].x = 240;
arc_point[9].y = 195;
arc_point[10].x = 240;
arc_point[10].y = 205;
arc_point[11].x = 220;
arc_point[11].y = 210;
arc_point[12].x = 210;
arc_point[12].y = 215;
arc_point[13].x = 210;
arc_point[13].y = 225;
arc_point[14].x = 220;
arc_point[14].y = 229;

```



```

GpiPolyFillet(hps,                /* Draw right side of base    */
              15L,
              &arc_point[0]
              );
GpiEndArea(hps);
GpiCloseSegment(hps);
GpiAssociate(hps,hwndClientDC);
GpiDrawSegment(hps,1L);

```

In this example, unassociated normal PS is created and the drawing mode is set to retained mode. The background color is then set to blue. A segment with an ID of 1 is opened and the picture of a lamp with a white shade and a brown base is drawn into it. The segment is then closed. At this point, you have a retained segment whose identifier is 1. To display the segment, the presentation space is associated with the window device context. GpiDrawSegment is then used to draw the segment just created. So, whenever you wish to use the picture of a lamp again, all you have to do is issue the GpiDrawSegment call.

When a segment is created, one piece of information specified is a *segment identifier*. This identifier is what makes each segment unique. The segment identifier can be any number from 0 to 16378.

Segments, like C functions, cannot be nested, but you can call a segment from within another segment. As a part of the segment-calling capability, it is possible to control the size, orientation, and placement of the picture created by the called segment. These features are provided by the transformation matrix parameter of the GpiCallSegmentMatrix call. Transforms are discussed in more detail later in this section.

Here is the way that segments may call other segments. One segment calls one or more other segments used to complete the picture. To demonstrate this process, Listing 19.2 creates another segment that draws a simple table and uses the lamp from segment one. To use that lamp within the other segment this new segment will call segment one.

Listing 19.2

```

GpiSetSegmentAttrs(hsPS,1L,ATTR_CHAINED, ATTR_OFF);
GpiSetDrawingMode(hps,DML_RETAIN); /* Set the drawing mode to retain */
GpiSetBackColor(hps,CLR_BLUE);    /* Set segment background to blue */
GpiOpenSegment(hps,2L);            /* Create segment 2                */
/*****
/* Table top definition.
/*****
GpiSetColor(hps,CLR_GREEN);
GpiBeginArea(hps,
              BA_ALTERNATE | BA_BOUNDARY
              );
array_point[0].x = 100;
array_point[0].y = 130;
array_point[1].x = 230;

```

```
array_point[1].y = 160;
array_point[2].x = 290;
array_point[2].y = 100;
array_point[3].x = 150;
array_point[3].y = 60;
GpiMove(hps,&array_point[3]);
GpiPolyLine(hps, 4L, &array_point[0]);
GpiEndArea(hps);
GpiSetColor(hps,CLR_DARKGREEN);
GpiBeginArea(hps,
    BA_ALTERNATE | BA_BOUNDARY
);
GpiMove(hps, &array_point[0]);
array_point[0].x = 100;
array_point[0].y = 60;
array_point[1].x = 110;
array_point[1].y = 50;
array_point[2].x = 120;
array_point[2].y = 60;
array_point[3].x = 120;
array_point[3].y = 85;
array_point[4].x = 140;
array_point[4].y = 50;
array_point[5].x = 140;
array_point[5].y = 10;
array_point[6].x = 150;
array_point[6].y = 0;
array_point[7].x = 160;
array_point[7].y = 10;
array_point[8].x = 160;
array_point[8].y = 50;
array_point[9].x = 270;
array_point[9].y = 78;
array_point[10].x = 270;
array_point[10].y = 40;
array_point[11].x = 280;
array_point[11].y = 30;
array_point[12].x = 290;
array_point[12].y = 40;
array_point[13].x = 290;
array_point[13].y = 100;
array_point[14].x = 150;
array_point[14].y = 60;
array_point[15].x = 100;
array_point[15].y = 130;
GpiPolyLine(hps, 16L, &array_point[0]);
GpiEndArea(hps);
```

```

/*****
/* Draw the lamp.
*****/
segmat.fxM11 = 0x00008000;
segmat.fxM12 = 0x00000000;
segmat.lM13 = 0x00000000;
segmat.fxM21 = 0x00000000;
segmat.fxM22 = 0x00008000;
segmat.lM23 = 0x00000000;
segmat.lM31 = 0x00000070;
segmat.lM32 = 0x00000050;
segmat.lM33 = 0x00000001;
GpiCallSegmentMatrix(hps,          /* Draw the lamp
                                1L,
                                0L,
                                &segmat,
                                TRANSFORM_REPLACE
                                );
GpiCloseSegment(hps);
GpiDrawChain(hps);

```

Editing a Segment

Now that you know how to create segments and call them from other segments, take a look at what is involved in editing a segment.

You already know that a segment contains the graphics primitives and attributes used to create a picture. Each primitive or attribute generates something called a *graphics order*, which is an unassembled assembler instruction form of the GPI API. When an order is placed in a segment, it becomes an element of the segment. There are two types of elements: a simple element, which is a graphic order, and a compound element, which is made up of two or more graphic orders.

A compound element is provided so you can treat a group of orders the same as you would treat a single order. A compound element is defined using the `GpiBeginElement` and `GpiEndElement` APIs. The orders that follow the “begin” function become a part of that element. The definition of the compound element is ended by using `GpiEndElement`. Between these calls is the definition of the element.

Inside these elements are the primitives that make up the elements, and the elements make up a segment.

If you know you will change specific parts of your segment, you can use a label to mark that point in the segment. A label is nothing more than a place holder that is an index into a segment for a particular element. A label is a four-byte integer that is assigned using the `GpiLabel` call. Labels are used to quickly identify an element within

a segment; you don't have to know either the exact offset of the element or how to interrogate the order to determine the element.

You can have duplicate labels within a segment. At times this is desirable. For example, say you are using a repeated sequence within a segment. Duplicate labels make it easier to find all the occurrences of the elements of interest, because they will all have the same label. Labels also come in handy when you are working with compound elements. The label is placed just before you issue the beginning element call, because labels cannot be a part of the compound element definition.

An example of label usage follows. Suppose you want to change the color of the lamp shade. The definition of segment one has been modified to add the labels as shown:

```
GpiSetDrawingMode(hps,DM_RETAIN); /* Set the drawing mode to retain */
GpiSetBackColor(hps,CLR_BLUE);    /* Set segment background to blue */
GpiOpenSegment(hps,1L);            /* Open segment 1 */
cur_pos.x = 180;
cur_pos.y = 300;
GpiMove(hps, &cur_pos);            /* Set current position (200,300) */
GpiSetColor(hps,CLR_BROWN);
arc_point[0].x = 200;
arc_point[0].y = 310;
arc_point[1].x = 220;
arc_point[1].y = 300;
GpiPolyFillet(hps,                /* Draw top of lamp */
              2L,
              &arc_point[0]
            );
/*****
/* Lamp shade definition.
*****/
GpiLabel(hps,3L);                  /* Label of three */
GpiSetColor(hps,CLR_WHITE);        /* Set shade color to white */
GpiBeginArea(hps,
              BA_ALTERNATE ! BA_BOUNDARY
            );
cur_pos.x = 140;
cur_pos.y = 300;
GpiMove(hps, &cur_pos);
cur_pos.x = 260;
GpiLine(hps,&cur_pos);              /* Draw top of shade */
cur_pos.x = 320;
cur_pos.y = 230;
GpiLine(hps,&cur_pos);              /* Draw right side of shade */
cur_pos.x = 80;
GpiLine(hps,&cur_pos);              /* Draw base of shade */
cur_pos.x = 140;
cur_pos.y = 300;
```

```

GpiLine(hps,&cur_pos);          /* Draw left side of shade      */
GpiEndArea(hps);
GpiLabel(hps, 4L);              /* Label of four          */
/*****
/* Lamp base definition.
*****/
GpiSetColor(hps,CLR_BROWN);     /* Set shade color to brown */
GpiBeginArea(hps,
              BA_ALTERNATE : BA_BOUNDARY
              );
cur_pos.x = 220;
cur_pos.y = 229;
GpiMove(hps, &cur_pos);
cur_pos.x = 180;
GpiLine(hps, &cur_pos);
arc_point[0].x = 190;
arc_point[0].y = 225;
arc_point[1].x = 190;
arc_point[1].y = 215;
arc_point[2].x = 180;
arc_point[2].y = 210;
arc_point[3].x = 160;
arc_point[3].y = 205;
arc_point[4].x = 160;
arc_point[4].y = 195;
arc_point[5].x = 180;
arc_point[5].y = 190;
arc_point[6].x = 190;
arc_point[6].y = 175;
arc_point[7].x = 180;
arc_point[7].y = 160;
arc_point[8].x = 140;
arc_point[8].y = 155;
arc_point[9].x = 140;
arc_point[9].y = 135;
arc_point[10].x = 180;
arc_point[10].y = 130;
arc_point[11].x = 180;
arc_point[11].y = 115;
arc_point[12].x = 160;
arc_point[12].y = 100;
arc_point[13].x = 120;
arc_point[13].y = 100;
arc_point[14].x = 100;
arc_point[14].y = 80;
GpiPolyFillet(hps,              /* Draw left side of base  */
              15L,
              &arc_point[0]

```

```

    );
    cur_pos.x = 300;
    cur_pos.y = 80;
    GpiLine(hps, &cur_pos);          /* Draw baseline of base */
    arc_point[0].x = 280;
    arc_point[0].y = 100;
    arc_point[1].x = 240;
    arc_point[1].y = 100;
    arc_point[2].x = 220;
    arc_point[2].y = 115;
    arc_point[3].x = 220;
    arc_point[3].y = 130;
    arc_point[4].x = 260;
    arc_point[4].y = 135;
    arc_point[5].x = 260;
    arc_point[5].y = 155;
    arc_point[6].x = 220;
    arc_point[6].y = 160;
    arc_point[7].x = 210;
    arc_point[7].y = 175;
    arc_point[8].x = 220;
    arc_point[8].y = 190;
    arc_point[9].x = 240;
    arc_point[9].y = 195;
    arc_point[10].x = 240;
    arc_point[10].y = 205;
    arc_point[11].x = 220;
    arc_point[11].y = 210;
    arc_point[12].x = 210;
    arc_point[12].y = 215;
    arc_point[13].x = 210;
    arc_point[13].y = 225;
    arc_point[14].x = 220;
    arc_point[14].y = 229;
    GpiPolyFillet(hps,                /* Draw right side of base */
        15L,
        &arc_point[0]
    );
    GpiEndArea(hps);
    GpiCloseSegment(hps);
    GpiAssociate(hps, hwndClientDC);
    GpiDrawSegment(hps, 1L);

```

Before a segment can be modified, you specify the type of editing to be done. The system must be notified if you are inserting new elements in the segment or if elements are to be replaced.

In the following coding example, the system is told that elements will be replaced. Once the edit mode is established, segment editing may begin. Editing takes place at

the location referenced by the current element pointer plus an offset. The initial element pointer and offset are both zero. The setting of the element pointer is accomplished using either the `GpiSetElementPointer` call or the `GpiSetElementPointerAtLabel` call. You can see where the label comes into play.

The element offset is then established using the `GpiOffsetElementPointer` call. The color of the lamp shade is then changed from white to pink. The system is then informed that an element is to be inserted, and the program inserts a `GpiCharString` element.

```
GpiSetEditMode(hps,          /* Set replace edit mode      */
               SEGMEM_REPLACE
               );
GpiSetElementPointerAtLabel(hps, /* Set element pointer to label */
                           3L /* three */
                           );
GpiOffsetElementPointer(hps, /* Point to the statement you want */
                       1L /* to replace */
                       );
GpiSetColor(hps, CLR_PINK); /* Make the shade pink */
GpiSetEditMode(hps,          /* Set insert edit mode      */
               SEGMEM_INSERT
               );
GpiSetElementPointerAtLabel(hps, /* Set element pointer to label */
                           4L /* four */
                           );

cur_pos.x = 160;
cur_pos.y = 270;
GpiCharString(hps, 6L, "Edited");
```

Copying Segment Data

In addition to editing a segment, you can create one that is a combination of two or more segments. Building a composite segment is accomplished by copying elements from the other segments into a new segment.

Element copying is accomplished with the `GpiGetData` call. The copied data is placed into the new segment with the `GpiPutData` API. First, you create the new segment. The data is then copied from the source segment to the new segment. Data may be retrieved from any random point within a segment, but when data is being placed into a segment, it may only be appended.

Segment Attributes

There are several types of retained segments. The type of retained segment is determined by the segment attribute. The attributes are chained, fast chained, detectable, dynamic, visible, propagate detectable, and propagate visible.

The propagate attributes state that if segment A is called by segment B and the propagate attribute is set for segment B but not for segment A, A inherits the attribute from segment B.

One of the attributes for a segment is "chained." The chaining of segments is nothing more than linking them. When a segment is created, the chained attribute is automatically set. With this attribute, segments created as part of the chain can be drawn as a single group. When a segment is created with the chained attribute in effect, it becomes a part of the segment chain for that presentation space. There can be only one segment chain in effect in a single presentation space. The segment chain may be manipulated by calling `GpiDrawChain`. A portion of the chain may be drawn using `GpiDrawFrom`.

You can remove segments from a chain as well as add them. If it becomes necessary to remove a segment or to create a segment that is not part of the chain for the presentation space, the chained attribute can be turned off. This is accomplished with a call to `GpiSetSegmentAttrs`.

Unchained segments are always retained. However, they are never drawn when the drawing mode is set to either draw or draw-and-retain. One reason to have unchained segments is that the segments can be used to form other segments via the `GpiCallSegmentMatrix` API.

When segment drawing begins, primitives' attributes can be reset to their default values or they may remain the way they are currently set. This is accomplished by using the "fast chaining" attribute. This attribute is turned on by default so that the primitive attributes remain the way they are currently set.

The detectable and the dynamic attributes are useful when a picture is being created and items that make up the picture can be selected and moved. In the example of the lamp and the table, it would be beneficial to turn on the detectable and dynamic attributes. The drawing control is set to `DCTL_DYNAMIC` in this case. This enables the user to place the mouse pointer on either the lamp or the table, press and hold button one on the mouse, and move the pointer (and the selection) to a new location in the window. This is an example of how the Presentation Manager interface provides powerful graphics support. Moving complex objects is now as simple as specifying an attribute in a segment.

The visibility attribute is similar to the `WS_VISIBLE` style in windows. When a piece of a picture is to be hidden, all that need be done is to turn off the visibility attribute. The default value of this attribute is that the segment is visible. If you make it invisible, there is no effect on the current position or attributes that would be changed when the segment is visible. These changes still occur; they are just not visible as they are occurring.

Summary

When you work with segments, both retained mode and draw and retain mode are available to you. If you are going to edit segments, you can only use retained mode. The use of segments enables you to break pictures up into manageable pieces. Each segment has its own attributes and may be placed in the window independently.

Using these functions, you can create pictures in pieces and arrange them however you like to create many different combinations of pictures from any number of segments.

Transforms

As mentioned in an earlier chapter, you can construct scaled drawings of life-size structures with the Presentation Manager Graphics Program Interface. The life-size structure, such as a house, is measured in feet and inches, much too large to fit in a Presentation Manager window. In such cases, a predefined scale is used. Such an example might scale 12 presentation page units to 1 foot to represent house measurements and preserve the scale. The scaled drawing of a house 150 by 80 feet is significantly smaller than the actual house. This drawing is, however, much larger than the size of the entire screen. What you really need is the ability to shrink this drawing to some manageable size. This capability is provided through transforms.

It is also easier to construct drawings if they are broken into components. Each component of the drawing represents a level of detail of the structure. The mechanism provided by the Presentation Manager to support this is graphics segments. The use of segments, along with transforms, gives applications substantial flexibility in managing complex pictures such as a house.

Figure

Coordinate Spaces

The use of transforms affords a great deal of flexibility to manipulate and manage a drawing. A *transform* is nothing more than a change applied to a picture. It is possible to reduce or enlarge the size of all or part of a drawing, change the position where an item is drawn, and rotate the item to an orientation of your choosing.

Up to now, you have seen how to create a presentation space, associate it with a device context, and issue drawing requests to the presentation space. This enables applications to draw pictures on the screen or printer paper without concern for the underlying representations and details of how that picture got there. These representations and details are handled automatically by the system.

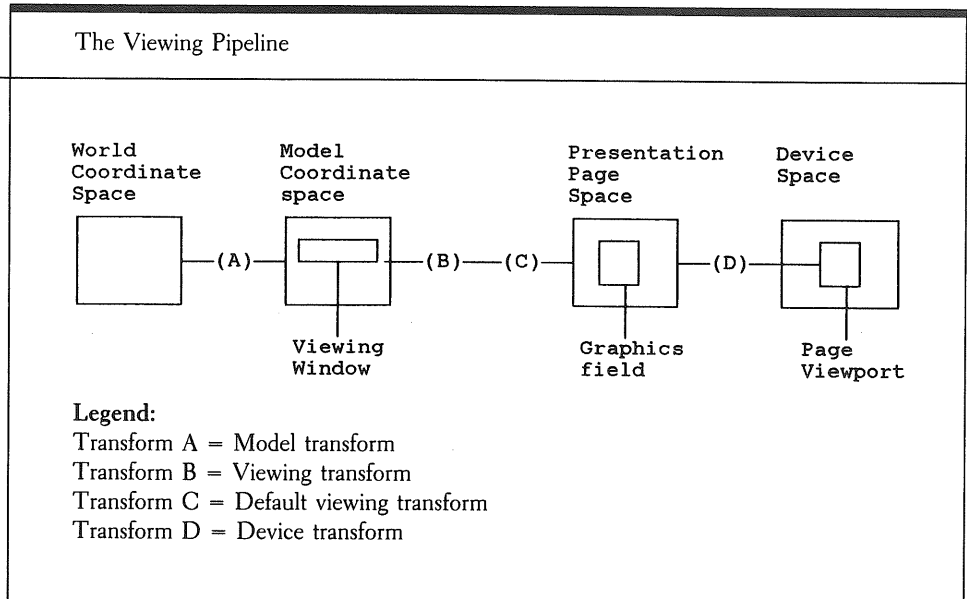
The representations are in terms of dimensional references called *coordinate spaces*. As you move from one coordinate space to another, a transform is applied. This enables programs to take very abstract information like the life-size representation of a building and transform it into something the system is capable of managing, like a scaled representation of the building. The representation transforms from a physical representation to a conceptual representation, and then to a physical representation.

The coordinate spaces are the world coordinate space, the model coordinate space, the presentation page space, and the device space. Figure 20.1 shows the viewing pipeline, the application of transforms from world coordinate space to device space, and the different parts discussed in this chapter.

The first coordinate space is called the “world coordinate” space. World coordinates are the values used when you create a component of a picture. They are presentation page coordinate values we use to represent some real relationship to the actual object being drawn. For example, using the arbitrary presentation page units, you can establish a scale where one unit represents one inch, one foot, or one meter.

These are the coordinate values used to create a segment that draws a window or a door. The valid range of values depends on the presentation page format you select,

Figure 20.1



which is either the long or short format. The short format ranges in values from -32768 to $+32767$ and the long format ranges in values from -134217728 to $+134217727$. The first transform that is applied to the drawing transforms from the world coordinate space to the model coordinate space.

This type of transformation is called a *model transform*. The model coordinate space is a conceptual rather than a physical coordinate space. It is considered conceptual because it is not something that can be created or referenced directly. It is possible, however, to control the appearance of a picture during this phase of its drawing using the different transforms. The model coordinate space is where the components of a picture are assembled so they form the complete picture. For example, the windows, doors, roof, and trim would be assembled to produce the complete picture of the house. The coordinate values have the same range as permitted by the presentation page format. The model transforms that are applied control the entire segment being manipulated, a part of the segment, or a single statement within a segment.

Segment Transforms

When you deal with a segment, you may want to control the entire contents of the segment with a single transform. In this case, a segment transform can be used. The segment transform is established using the `GpiSetSegmentTransformMatrix` function. This function replaces the transform matrix, so it is a good idea to save the previous transform values before you change them. This way they may be restored without a lot of fuss.

Saving the values is relatively simple. `GpiQuerySegmentTransformMatrix` gets the current transform values, which may then be saved for future use. Typically, the segment transform is set, then the `GpiDrawXXX` functions are executed to draw the segments that would use the new transform. After the segments have been drawn, the transform should be restored to its previous values.

Model Transforms

If only a part of a segment changes during the course of an application, use a transform that enables modification of some of the primitives in the segment. This type of transform is called the *model transform*.

Suppose the segment being manipulated draws a front door, and you want to change only the size of the knocker on the door. A model transform can be used to perform this function. The model transform is changed by the `GpiSetModelTransformMatrix` call.

The way you might change the knocker on the door is as follows: Within the door definition segment the current transform matrix is first queried using `GpiQuery-ModelTransformMatrix`, and the returned values are saved for later. `GpiSetModelTransformMatrix` then is called to change the transform matrix as desired, and then the affected part of the segment is drawn. In this example that part is the door knocker. Once this change is completed, the transform is returned to its prior settings by calling `GpiSetModelTransformMatrix` with the saved values.

A model transform can also be processed in another way. This method is most appropriate if an item is to be drawn repeatedly in different positions within a single segment or an item common to several segments is being used. This method enables you to define an item once and then use it multiple times in the same segment or in different segments. This type of transform is a *single-line transform* (also called an *instance transform*).

Instance transforms are applied by calling the `GpiCallSegmentMatrix` API. An example of an instance transform was shown in Chapter 19's discussion of graphics segments, Listing 19.2. The example applied the transform in the definition of the table segment. The instance transform was used to draw the lamp to reduce its size and move it to a corner of the table so that the lamp appears to be on the table in the distance.

Viewing Transforms

Now that the parts of the picture are assembled to produce a complete picture, another set of transforms is applied to take it from the model space to the presentation page space. This transform takes it back to a physical coordinate space. There are two transforms applied when you go from the model space to the presentation page space.

The first is the viewing transform. Viewing transforms are applied only to primitives within segments and have no effect on primitives outside segments. The viewing transforms apply only to segments that have been created since the transform has been established. When the need arises to size part of a picture or to scroll a picture, viewing transforms can be used.

An example of this usage is a user looking at a state map that displays very detailed information. The user might select a function that allows the city streets for a selected area on the map to be seen. Such a display could be easily handled in a single presentation page. The problem is that the entire presentation page is used. The procedure can be simplified by identifying the areas of interest and the area of the presentation page that should be used to display the new information by defining new bounding rectangles.

A viewing window first needs to be defined. What this is telling the system is that the entire state is not to be used, but just an area of the city that is below the point

of interest. This is accomplished using `GpiSetViewingWindow` to define the boundary of the area to be used from the model space.

Suppose it is not desirable to replace the map already drawn. It would be necessary to define a graphics field where the new information is to be displayed. The graphics field is a rectangular subset of the presentation page that will be updated by the new information. A graphics field that is equivalent to the upper quarter of the presentation page needs to be defined.

Once this is done, `GpiSetViewingTransform` is used to set the viewing transform to this new field. After the viewing transform is established, a segment is created to hold the definition for the area of interest. The segment is then called to be drawn. The reason for creating a segment on the fly is that the viewing transform only has effect on segments created after the transform has been established.

Device Space Transforms

The last type of transform translates from the presentation page space to device space. The *device space transform* is established when a presentation space is created. The transformation is a scaling (or sizing) transformation. The entire picture is always displayed.

In using device space transforms, the `GpiSetPageViewport` API is used to change the page viewport. The final coordinates after the transform must be between -32768 and $+32767$, regardless of the presentation page format used.

Transformation Matrix

The *transformation matrix* enables you to perform translation, scaling, and rotating transforms. There are nine elements in a transformation matrix. The third, sixth, and ninth elements are constants and have the values of 0, 0, 1, respectively. For this discussion, give the other elements the names A–F and refer to them as such. The matrix would look like (A, B, 0, C, D, 0, E, F, 1) and is often shown as

A, B, 0
C, D, 0
E, F, 1

The transform matrix

Translation

Translation occurs when an item is moved from one location to another. For example, if a box is defined with its lower left corner at coordinates (50,30), a translation transform

is used to move the box to a position where its lower left corner is at (90,80). The new x coordinate is determined by adding the old x coordinate to element E, and the new y coordinate is determined by adding the old y coordinate to element F.

In this example, E would have a value of 40 and F would have a value of 50 to produce the new coordinates of (90,80). If E is a positive value, the movement will be to the right and conversely to the left if E is a negative value. If F is a positive value, movement will be up; the movement is down if F is negative. Both elements E and F are LONG values.

Scaling

A *scaling transform* is used to change the size of an item. The size can be either increased or decreased. The matrix elements A and D are used for scaling. Both elements are fixed values. A fixed value contains a notational binary point between bytes two and three.

For example, a value of 1.0 is given as 0x00010000 (hexadecimal), and 1.45 is given as x00017333. If the scaling value used is greater than one, the size increases, and if values are less than one, the picture size is decreased.

When the transformation matrix is used, the new x coordinate is derived by multiplying the old x coordinate by element A in the matrix. The new y coordinate is derived by multiplying the old y coordinate by element D from the matrix. To make the lamp from the previous example, Listing 19.2, half as big, a scaling factor of one half, 0x00008000, is used for both elements A and D. When the lamp was drawn using this scaling value, it was displayed half as big as it was defined.

Rotation

The last type of the transform is rotational. A *rotational transform* allows a program to change the orientation of an item. An item can be rotated in either the clockwise or the counterclockwise direction. The matrix elements that are used for performing a rotation are A, B, C, and D, and they are all fixed.

When you rotate an element clockwise, the A element of the matrix is the cosine of the angle of rotation, element B is the negative sine of the angle, C is the sine of the angle, and D becomes the cosine of the angle.

The new x coordinates for the picture are obtained by multiplying the previous x coordinate by the element A, then adding the result of multiplying the previous y coordinate by element C. The new y coordinates are obtained by multiplying the previous x coordinate by element B, then adding the value derived by multiplying the previous y coordinate by element D.

When you use counterclockwise rotations, element A is the cosine of the angle, B is the sine of the angle, C is the negative sine of the angle, and D is the cosine of the angle.

The new coordinates are derived by using the exact same multiplication formula given above for clockwise rotations.

Summary

Transforms are a complex subject in the graphics world. Using the Presentation Manager drawing functions, performing transforms becomes simpler than on other systems. There are specific APIs to perform transforms, thereby eliminating most of the calculation work from your applications.

The OS/2 GPI provides transform matrices to perform the various types of transforms. Rotation, translation, and scaling may be performed singly or in combination using the OS/2 Presentation Manager GPI.

Storing and Printing Application Graphics

The graphics storage methods discussed thus far are considered temporary, because the image is destroyed when the program terminates. The Presentation Manager GPI offers several more powerful methods of storing graphics. One of these methods involves the use of graphic segments. Other storage methods include bitmaps, metafiles, and the clipboard.

Two of these methods—bitmaps and metafiles—also provide a more permanent method of storing graphics. Bitmaps and metafiles can be used to import graphics from other programs or to export graphics to other programs. The use of bitmaps and metafiles is discussed in this chapter. The clipboard is discussed in Section 6.

Bitmaps

A *bitmap* is composed of pels on the screen or dots on a printer. It is a means of creating images within programs. These images may represent a variety of items such as icons, pointers, and other two-dimensional pictures. A bitmap is composed of two parts: the bitmap header and the bitmap data. This section describes the components of bitmaps in detail.

Bitmap Header

The *bitmap header* provides the specifics for the bitmap. It contains five pieces of information:

- header length
- bitmap width
- bitmap height
- bitmap planes
- bitmap format

The header length specifies the number of bytes that make up the bitmap header. The width value specifies the number of pels that make up a single line of the bitmap.

Each line of a bitmap is called a *scan line*. The height value in the bitmap header indicates the number of scan lines in the bitmap.

A bitmap *bit plane* is the width of the bitmap times the height. Most bitmaps require no more than a single plane. A *multiple-plane bitmap* is one with two or more bit planes. The multiple-plane bitmap is provided to support devices that have multiple planes.

The bitmap format specifies the number of bits used to draw a single pel on each plane. The standard formats supported by the Presentation Manager are 1 bit per pel, 4 bits per pel, 8 bits per pel, and 24 bits per pel. The 24 bits per pel format is standard.

Bitmap Data

Immediately following the bitmap header information is the *bitmap data*. The data contains the information to display the bitmap. The data represents the image's scan lines from left to right and bottom to top. The internal representation of each scan line is padded on the right so that each scan line starts on a four-byte boundary.

Creating a Bitmap

A bitmap can be created in one of two ways: by an application or by the Icon Editor.

When you create a bitmap in an application program, the first step is to open a memory device context. You then create a presentation space and associate it with the memory device context. The bitmap information header is then initialized, and the `GpiCreateBitmap` function is called to create the bitmap. The handle that is returned for the bitmap is then used to "select" the bitmap into the presentation space. This is done with `GpiSetBitmap`.

Once the bitmap has been created and selected as current, data may now be placed into the bitmap. The data that is put in this new bitmap can come from several sources:

- copied from another bitmap
- interactive entries, as in an Icon Editor type application
- drawn using graphics primitives

Once these steps have been completed, the bitmap is considered to be in temporary storage. It may be subsequently permanently saved by writing the header information followed by the bitmap data to a data file. An example of this entire function appears in Listing 21.1.

Listing 21.1

```
DEVOPENSTRUCT device_open_data[9]={0L,
                                     "MEMORY",
                                     0L,
                                     0L,
                                     0L,
                                     0L,
                                     0L,
                                     0L,
                                     0L
                                   };

hmdc = DevOpenDC(hab,                /* Open memory DC          */
                 OD_MEMORY,
                 ".*",
                 4L,
                 (PDEVOPENDATA)&device_open_data,
                 NULL);

size.cx = 640;
size.cy = 480;                      /* Create bitmap PS        */
hmps = GpiCreatePS(hab,
                   hmdc,
                   &size,
                   GPIA_ASSOC ! GPIF_LONG !
                   GPIT_NORMAL ! PU_PELS
                   );
                                   /* Initialize bitmap header */
bmapinfo.cbFix = 12;
bmapinfo.cx = 200;
bmapinfo.cy = 150;
bmapinfo.cPlanes = 1L;
bmapinfo.cBitCount = 4L;
hbmap = GpiCreateBitmap(hmps,        /* Create a color bitmap   */
                        (PBITMAPINFOHEADER)&bmapinfo,
                        0L,
                        (PBYTE)NULL,
                        (PBITMAPINFO)NULL
                        );
GpiSetBitmap(hmps,                  /* Set bitmap as current bitmap */
             hbmap
             );
hps = WinGetPS(HWND_DESKTOP);
```

```

cpy_array[0].x = 0;
cpy_array[0].y = 0;
cpy_array[1].x = 200;
cpy_array[1].y = 150;
cpy_array[2].x = 0;
cpy_array[2].y = 0;
cpy_array[3].x = 200;
cpy_array[3].y = 150;
GpiBitBlt(hmps,                      /* Copy data to the bitmap */
          hps,
          4L,
          &cpy_array,
          ROP_SRCCOPY,
          BBO_OR
          );
cur_pos.x = 180;
cur_pos.y = 5;

GpiCharStringAt(hmps,                /* Place string in bitmap */
                &cur_pos,
                40L,
                "Sample Bitmap Created by an application."
                );

GpiBitBlt(hps,                      /* Display bitmap on screen */
          hmps,
          4L,
          &cpy_array,
          ROP_SRCCOPY,
          BBO_OR
          );

DosOpen("SCRCPY.BMP",                /* Open bitmap disk file */
        fhnd,
        2,
        0,
        0x0020,
        0x0012,
        0x4092,
        0L
        );

DosWrite(fhnd,                       /* Write header to the file */
         &bmapinfo,
         sizeof(BMAPINFOHEADER),
         &bytecnt
         );

for(indx = 0; indx < 150; indx++)
{
    GpiQueryBitmapBits(hmps,          /* Get bits from the bitmap */
                       indx,

```

```

        1L,
        bufptr,
        &bmapinfo,
        scan_cnt
    );
    if(scan_cnt > 0)
        DosWrite(fhnd,                /* Write bitmap data to file */
            bufptr,
            600,
            &bytecnt
        );
    }
    DosClose(fhnd);                    /* Close bitmap file */
    WinReleasePS(hps);                 /* Free the PS */

```

Listing 21.1 creates a bitmap and copies the lower 200×150 corner of the screen into the bitmap. Some text is then written into the bitmap. The bitmap is then drawn in the window to verify the results. It is then permanently saved in a file called SCRCPY.BMP. That bitmap is now available for use by any program and is not destroyed when the application terminates.

Loading a Bitmap

There are two ways to load a bitmap for an application program. The first uses a method that is almost exactly the reverse of the bitmap creation procedure. The file is opened, the header is read, the bitmap is set, and finally the data is transferred from the file to the bitmap.

The other method for loading a bitmap is to include it as a part of the application's resources and to use `GpiLoadBitmap` to load it into memory. Once the bitmap has been loaded, `GpiSetBitmap` is used to select the bitmap.

You can create either monochrome or color bitmaps inside an application or using the Icon Editor. However, the Icon Editor is limited to creating single-plane bitmaps. The size of the bitmap that can be created with the Icon Editor depends on the type of display you have and the type of bitmap you are creating; application-created bitmaps may be much larger.

A bitmap is a device-specific resource. The coordinates used to create a bitmap are device coordinates. For example, a bitmap created for use on a display with square pixels, such as a VGA display, does not look the same when displayed on an EGA monitor. The EGA version looks more rectangular. The effect of the adapter card in the user's machine is something to be aware of when you create bitmaps. The primary benefit of a bitmap is the speed with which it can be drawn or removed from the screen. Although it can be quickly drawn, the bitmap does not offer the editing flexibility of a graphics segment. If you elect to modify a bitmap, you do so one bit at a time.

Metafiles

Importing and exporting graphics images is a desirable feature of graphics applications. The emphasis is on being able to produce or obtain device-independent graphics. The Presentation Manager Graphics Program Interface (GPI) supports the temporary storage of device-independent graphics through graphics segments, and it supports the permanent storage of device-independent graphics through metafiles. Bitmaps are more device dependent than are metafiles.

You create a metafile in three stages. First is the definition stage. This is the stage in which you create the device context, add the graphics orders that make up the metafile, and close the device context. This definition stage begins when the program uses the `DevOpenDC` function to open a metafile device context. This step returns the metafile device context handle that is associated with a presentation space. Once you define the metafile, use the graphics APIs to place the primitives in the metafile. Then use `DevCloseDC` to close the metafile.

Once the device context is closed, no more graphics may be added to the metafile. When `DevCloseDC` is called, a metafile handle is returned. The metafile now exists as a memory file and is referred to as being in the "memory file" stage.

The handle just returned is used for all future references to the metafile. During the memory file stage, the metafile is just another temporary means of storing graphics. The contents of the metafile can be displayed by calling the `GpiPlayMetafile` API. The metafile may be edited with `GpiSetMetafileBits`. The final stage of metafile creation is saving the memory file to disk.

To write the metafile to disk, call `GpiSaveMetafile`. When this is done, the metafile is also deleted from memory and the metafile handle being used is invalidated. You now have a permanent copy of the metafile that can be exported to other applications. The program code required to accomplish this is

```
DEVOPENSTRUCT  device_open_data[9]={0L,
                                     "DISPLAY",
                                     0L,
                                     0L,
                                     0L,
                                     0L,
                                     0L,
                                     0L,
                                     0L
                                     };

HMF  hmfdc;

.
.
hmfdc = DevOpenDC(hab,
                 OD_METAFILE,
```

```

        "x",
        4L,
        (PDEVOPENDATA)&device_open_data,
        NULL
    );

    size.cx = 640;
    size.cy = 480;
    GpiCreatePS(hmfps,
        hmfdc,
        &size,
        GPIA_ASSOC | GPIF_LONG |
        GPIT_MICRO | PU_PELS
    );

    cur_pos.x = 50;
    cur_pos.y = 150;
    GpiCharStringAt(hmfps,
        &cur_pos,
        25L,
        "Start of sample metafile."
    );

    /* .....

        .
        .
        .
        .

    Add the other graphics calls that will define the metafile here.

        .
        .
        .
        .

    ..... */
    cur_pos.x = 50;
    cur_pos.y = 10;
    GpiCharStringAt(hmfps,
        &cur_pos,
        22L,
        "End of sample metafile."
    );
    hmfdc = DevCloseDC(hmfdc);          /* Close DC get metafile handle */
    GpiAssociate(hmfps, NULL);          /* Disassociate metafile DC */
    GpiAssociate(hmfps, hwndClient);    /* Associate client area */
    GpiPlayMetafile(hps,                /* Display metafile contents */
        hmfdc,
        OL,
        NULL,
        OL,

```

```

        OL,
        NULL
    );
    GpiSaveMetafile(hmfps,          /* Save metafile on disk      */
        "SAMPLEMF.MET"
    );
    GpiAssociate(hmfps, NULL);      /* Disassociate window DC  */

```

Using an Existing Metafile

You may design applications that import graphics from other applications. Importing graphics is accomplished by calling `GpiLoadMetafile` to make the graphics in a metafile available to your application. The system returns a metafile handle that is used to reference the metafile. The graphics imported can be displayed in the same way as any metafile you create. `GpiPlayMetafile` displays any metafile, given the proper metafile handle. Once the application is through using the metafile, `GpiDeleteMetafile` can be called to unload the metafile, as demonstrated in the following example:

```

hmfdc = GpiLoadMetafile(hmfps,    /* Save metafile on disk      */
    "SAMPLEMF.MET"
);
GpiAssociate(hmfps, hwndClient);  /* Associate window DC        */
GpiPlayMetafile(hps,             /* Display metafile contents  */
    hmfdc,
    OL,
    NULL,
    OL,
    OL,
    NULL
);
GpiAssociate(hmfps, NULL);        /* Disassociate window DC     */
GpiDeleteMetafile(hmfps,
    hmfdc
);

```

Printing Graphics

As a part of your application design, you will provide a way for your application users to produce hardcopy results of their work. The type of support chosen depends on the type of display being created. If your application is to be text only, base system functions (or C runtime functions) can be used to provide this support. If you are creating a graphics application, Presentation Manager API functions are needed to provide printing capabilities.

Simple Text Printing

The base system calls provide simple printouts using the system default font. The following example shows simple text printing. DosOpen opens the printer device. DosWrite sends the device the data to be printed. The handle returned from DosOpen is used by DosWrite to direct the output to the proper place (the printer). When the program is finished sending data to the printer, DosClose closes the connection to the printer. If the OS/2 Print Spooler is running, the data is spooled, and when you issue DosClose the data is sent to the printer.

```
short prthnd, action_taken, wrt_count;

/* Open the printer */
DosOpen("LPT1",
        &prthnd,
        &action_taken,
        0,
        0x0020,
        0x0011,
        0x0021,
        0);
/* Path name of file to be opened */
/* File handle */
/* Action taken */
/* File's new size */
/* File attribute */
/* Action taken if file exists */
/* Open mode */
/* Reserved (must be 0) */

/* Write data to printer */
DosWrite(prthnd,
        "Hello World",
        11,
        &wrt_count);
/* File handle */
/* Data to print */
/* Buffer length */
/* Data written */

DosClose(prthnd);
/* Close printer */
```

Direct Printing

The Presentation Manager API functions can print either text or graphics. You can print directly to the printer or use the print spooler. When the Presentation Manager printing functions are used as opposed to the base functions, some flexibility in the types of fonts available is gained.

The preferred method is to use the spooler. For completeness, both methods are shown in this chapter. In the example that follows, the printer is opened for direct printing and the font is changed from the system default font to the Helvetica font.

```
FATTRS fonsel;
USHORT index;
SIZEL size;
HAB hab;
/* Font attribute structure */
/* Search index variable */
/* Size of PS variable */
/* Anchor block handle */
```

STORING AND PRINTING APPLICATION GRAPHICS

```

HDC hdpDC; /* Direct printer device context */
LONG lcid; /* Local identifier variable */
LONG FontCount; /* Define the font count variable */
PFONTMETRIC fontmet; /* Fontmetric pointer */
LONG ReqFont; /* Define the requested font */
/* Count variable */
DEVOPENSTRUC prt_direct[9]={ "LPT1", /* Logical device addr ptr */
                              "IBM4201", /* Device driver name ptr */
                              OL, /* Device data ptr */
                              OL, /* Queue file data ptr */
                              OL, /* Spool comment ptr */
                              OL, /* Queue processor name ptr */
                              OL, /* Queue proc parm ptr */
                              OL, /* Spooler parameter ptr */
                              OL }; /* Network parm ptr */

.
.

hdpDC = DevOpenDC(hab, /* Open printer DC */
                  (LONG)OD_DIRECT,
                  (PSZ)"*",
                  (LONG)4,
                  (PDEVOPENDATA)&prt_direct[0],
                  (HDC)0
                  );

size.cx = 0;
size.cy = 0;
hps = GpiCreatePS(hab, /* Create and associate PS */
                  hdpDC,
                  &size,
                  GPIA_ASSOC | GPIF_LONG |
                  PU_PELS | GPIT_NORMAL
                  );
/* Load the Helvetica font */
GpiLoadFonts(hab,
              "C:\\OS2\\DLL\\HELV.FON");

FontCount = GpiQueryFonts(hps,
                           QF_PUBLIC | /* Select public and */
                           QF_PRIVATE, /* private fonts */
                           NULL, /* Query all available fonts */
                           NULL, /* Information not */
                           NULL /* requested */
                           );

if (FontCount == NULL)
{
    /* Allocate font count times the */
    /* font metrics structure size */

```

```

    DosAllocSeg((FontCount * sizeof(FONTMETRICS)),
                &(SELECTOROF(fontmet)),
                SEG_NONSHARED);
}

if (fontmet != NULL)
{
    ReqFont = FontCount;
    FontCount = GpiQueryFonts(hps,
                              QF_PUBLIC |      /* Select public and      */
                              QF_PRIVATE,      /* private fonts         */
                              NULL,            /* Query all available fonts */
                              ReqFont,         /* Number of fonts requested */
                              sizeof(FONTMETRICS), /* Size of each metric    */
                              fontmet          /* Requested font metrics buffer */
                              );

                                /* Font selection logic      */
    for (index = 0; index < ReqFont; index++)
    {
                                /* Check for and image font      */
        if (!(fontmet[index].fsdefn & 0x8000))
        {
                                /* Check for a 6 x 9 character      */
            if (fontmet[index].lAveCharWidth == 6L &&
                fontmet[index].lMaxBaselineExt == 9L)
            {
                                /* Specify FATTR size          */
                fontset.usRecordLength = sizeof(FONTMETRICS);
                                /* Save the match number          */
                fontset.lMatch = fontmet[index].lMatch;
                                /* Save the face name            */
                sprintf(fontset.szFacename, fontmet[index].szFacename);
                                /* Save codepage                */
                fontset.usCodepage = fontmet[index].usCodepage;
                index = ReqFont;    /* End the search          */
            }
        }
    }

    DevEscape(hpdDC,            /* Specify document start */
              DEVEESC_STARTDOC,
              0L,
              NULL,
              NULL,
              NULL
              );

    lcid = 20;                  /* Initialize local ID     */
}

```

```

rc = GpiCreateLogFont(hps,
                      "MyLogFnt", /* Logical font name      */
                      lcid,      /* Local ID for this font */
                      &fontsel); /* Font attribute structure */

if (rc == 2L)
    GpiSetCharSet(hps, lcid); /* Establish current font */

DevEscape(hpdDC, /* Specify start of new page */
          DEVESC_NEWFRAME,
          0L,
          NULL,
          NULL,
          NULL
        );

cur_pos.x = 10;
cur_pos.y = 10;
GpiCharStringAt(hnPS, /* Print some text starting at */
                &point, /* current position (10,10)    */
                22L,
                "This is a direct print");

/***** Other text and graphics calls may be added here *****/

DevEscape(hdpDC /* Specify end of document */
          DEVESC_ENDDOC,
          0L,
          "",
          NULL,
          NULL
        );

GpiAssociate(hps, /* Disassociate PS */
             (HDC) NULL);

DevCloseDC(hdpDC); /* Close printer DC */

```

This example starts by opening a printer device context and associating it with a presentation space. The font is then changed from the system font to Helvetica, and the text is sent to the printer. Although the initialization of the font is done before the document is started, the creation and setting of the logical font must be executed after the DevEscape function is used to start the document. If the logical font creation was attempted before the document was started, an error would have occurred.

Queued Printing

The following example demonstrates writing to a printer that is being governed by the OS/2 Print Spooler. This type of printing is also referred to as *queued writing*, because each file printed is added to a "line" of "print jobs" waiting for their turn to print.

Listing 21.2 demonstrates how to print a screen or window.

Listing 21.2

```

BITMAPINFOHEADER bmapinfo;          /* Bitmap information header */
POINTL cpy_array[4];                /* Bitmap arrays */
SIZEL size;                          /* Size of PS variable */
HAB hab;                             /* Anchor block handle */
HDC hmdc;                            /* Memory device context */
HDC hqpdc;                           /* Queued printer device context */
HPS https;                           /* Temporary PS handle */
HPS hqps;                             /* Permanent printer PS handle */
HPS hps;                              /* Permanent bitmap PS handle */
DEVOPENSTRUC prt_queued[9]={ "LPT1Q", /* Logical device addr ptr */
                             "IBM4201", /* Device driver name ptr */
                             OL,        /* Device data ptr */
                             OL,        /* Queue file data ptr */
                             OL,        /* Spool comment ptr */
                             OL,        /* Queue processor name ptr */
                             OL,        /* Queue proc parm ptr */
                             OL,        /* Spooler parameter ptr */
                             OL };      /* Network parm ptr */
DEVOPENSTRUC prt_direct[9]={ "LPT1",   /* Logical device addr ptr */
                             "IBM4201", /* Device driver name ptr */
                             OL,        /* Device data ptr */
                             OL,        /* Queue file data ptr */
                             OL,        /* Spool comment ptr */
                             OL,        /* Queue processor name ptr */
                             OL,        /* Queue proc parm ptr */
                             OL,        /* Spooler parameter ptr */
                             OL };      /* Network parm ptr */

hmdc = DevOpenDC(hab,              /* Open memory DC for display */
                 OD_MEMORY,
                 ".*",
                 OL,
                 NULL
                 (HDC)0);

size.cx = 640;
size.cy = 480;
hps = GpiCreatePS(hab,
                  hmdc,
                  &size,
                  GPIA_ASSOC | GPIF_LONG |
                  GPIT_NORMAL | PU_PELS
                  );

bmapinfo.cbFix = 12;
bmapinfo.cx = 640;
bmapinfo.cy = 480;
bmapinfo.cPlanes = 1L;

```

```

bmapinfo.cBitCount = 1L;
hbmmap = GpiCreateBitmap(hps,          /* Create monochrome bitmap */
                        (PBITMAPINFOHEADER)&bmapinfo,
                        0L,
                        (PBYTE)NULL,
                        (PBITMAPINFO)NULL
                        );
GpiSetBitmap(hps,                      /* Set the bitmap as active */
             hbmmap
             );
https = WinGetPS(HWND_DESKTOP);        /* Get screen PS          */
cpy_array[0].x = 0;
cpy_array[0].y = 0;
cpy_array[1].x = 640;
cpy_array[1].y = 480;
cpy_array[2].x = 0;
cpy_array[2].y = 0;
cpy_array[3].x = 640;
cpy_array[3].y = 480;
GpiBitBlt(hps,                        /* Copy screen to bitmap  */
          https,
          4L,
          &cpy_array[0],
          ROP_SRCCOPY,
          BBO_OR
          );
WinReleasePS(https);                  /* Release screen PS      */
GpiSetBitmap(hps,                      /* Deselect the bitmap    */
             NULL);
GpiAssociate(hps,                      /* Disassociate the PS    */
             NULL);

DevCloseDC(hmdc);                     /* Close screen memory DC */
hqpDC = DevOpenDC(hab,                 /* Open printer DC        */
                  (LONG)OD_QUEUED,
                  (PSZ)"*",
                  (LONG)4,
                  (PDEVOPENDATA)&prt_queued[0],
                  (HDC)0
                  );

size.cx = 0;
size.cy = 0;
hqps = GpiCreatePS(hAB,                /* Create printer PS      */
                  hqpDC,
                  &size,
                  GPIA_ASSOC | GPIF_LONG |
                  GPIT_NORMAL | PU_PELS
                  );

```

```

hmdc=DevOpenDC(hab,                                /* Open memory DC for printer */
               OD_MEMORY,
               "*",
               OL,
               NULL,
               hqpDC);

GpiAssociate(hps,                                   /* Associate memory DC      */
             hmdc);

GpiSetBitmap(hps,                                   /* Set the bitmap as active */
             hmdc);

DevEscape(hqpDC,                                    /* Specify document start   */
          DEVEscape_STARTDOC,
          OL,
          "",
          NULL,
          NULL
        );

cur_pos.x = 10;
cur_pos.y = 10;
GpiCharStringAt(hqps,                               /* Print text starting at   */
                &point,                             /* current position (10,10) */
                22L,
                "Print screen example ");

DevEscape(hqpDC,                                    /* Specify start of new page */
          DEVEscape_NEWFRAME,
          OL,
          NULL,
          NULL,
          NULL
        );

cpy_array[1].x=960;
cpy_array[1].y=720;
GpiBitBlt(hqps,                                     /* Print bitmap             */
          hbmap,
          4L,
          &cpy_array,
          ROP_SRCCOPY,
          BBO_OR
        );

DevEscape(hqpDC,                                    /* Specify end of document  */
          DEVEscape_ENDDOC,
          OL,
          "",
          NULL,

```

```

        NULL
    );
    GpiAssociate(hqps,                /* Disassociate printer PS */
        (HDC)NULL);
    GpiSetBitmap(hps,                /* Deselect bitmap */
        NULL);
    GpiAssociate(hps,                /* Disassociate memory PS */
        (HDC)NULL);
    GpiDestroyPS(hps);              /* Destroy memory PS */
    GpiDestroyPS(hqps);             /* Destroy printer PS */
    DevCloseDC(hmdc);               /* Close memory DC */
    GpiDeleteBitmap(hbmap);          /* Delete the bitmap */
    DevCloseDC(hqpDC);              /* Close printer DC */

```

This example demonstrates printing graphics and text together. It demonstrates how a print screen or window print is done in the Presentation Manager session. The same procedure could also be done with direct printing. The screen content or window content is copied to a bitmap. The bitmap is selected into a printer memory device context and then printed by doing a bit blit. Other text or graphics that is not already in a window can be printed by sending the information directly to the presentation space associated with the printer's device context.

Summary

OS/2 is capable of storing many types of data. Among these types are specially defined data formats for graphics. Using these formats, storing and redisplaying stored graphics is a simple task. Bitmaps, metafiles, and icons are among the graphics formats you can store in data files and repaint at any time.

When you store graphics, you can create a history of data or pictures. They may be archived or called up at a moment's notice.

In many instances, you provide a means of generating hard copy output so your users can print their work. The GPI provides facilities to print graphics to any supported hardcopy device. This can range from simple, dot-matrix printers to complex graphics plotters.

Using the device-independent features of the Presentation Manager, graphics may be output in hard copy just as easily as painting them in a window.

SECTION SIX

ADVANCED TOPICS

The previous sections presented a set of fundamentals that laid the groundwork for any application you may wish to write.

This section contains tips and techniques for using some of the advanced functions available for Presentation Manager programs. Included here are chapters dealing with dynamic link libraries, application debugging, and exchanging data among programs.

Application Data Transfer

Transferring data easily from one application to another has long been an unmet need of application users. Operating systems in the past have neglected to provide this service as a part of the API for applications to use.

Early DOS applications relied on disk files for transferring data from one application to another. Pipes were provided by DOS as a system feature, but there was no API for applications to use that feature. OS/2 provides pipes as a part of the system services, as well as a part of the API for application use.

A *pipe* is a “pseudofile” mechanism used to pass data sequentially. This mechanism offered a little improvement over disk files, because under OS/2, both applications can be active when the data is transferred.

OS/2 provides a significant improvement for transferring and sharing data through *shared memory*. This mechanism allows two or more applications to have concurrent access to a previously allocated piece of memory. This memory can be greater than 64K and can be accessed sequentially or randomly. It is the application’s responsibility to ensure data integrity. These are features of the base operating system. The Presentation Manager mechanism provides support for a similar interface, but the operating system ensures data integrity. This support is provided through the clipboard interface. Another means of interprocess communication provided by Presentation Manager is Dynamic Data Exchange (DDE). DDE allows applications to establish dialogs when the programs transfer data. This chapter first discusses the use of the clipboard interface by applications and then by DDE.

Clipboard

When the clipboard is mentioned, the first thing that might come to mind is a piece of memory of a fixed size that is maintained by the system. This is not the case. Using the clipboard is much like walking into a car rental office to pick up the keys for a car. At the time you walk in the door, you have an idea what type of car you want, but you are not sure what you will finally get. Once you get the keys for a car, you know the type of car you will drive. You have access to the car as long as you have the keys. Once you return the keys, you no longer have access to the car. So it is with the clipboard. Applications can determine whether there is data available and the format of the available data. The data can be used, but after the clipboard is closed, the application no longer has access to the clipboard.

The clipboard is a user-driven interface that has two levels of support: user level and application level. You provide the user-level support in your application. The system provides application support.

Access to the clipboard is sequential; one application gains access at a time. The format of the data is determined by the application that placed the data in the clipboard. That application also determines the size of the data. If you support the clipboard services in your application, three functions are available: Cut, Copy, and Paste.

The reason the clipboard is called a user event-driven interface is because data is not transferred to or from the clipboard unless the user requests it. The user relies on visual feedback from the application to effectively use the clipboard.

Applications must make users aware that clipboard services are supported and provide feedback for data selection. The nonvisual functions involve placing data in the clipboard and retrieving data from the clipboard. Clipboard service is always indicated by placing the Cut, Copy, or Paste options on a menu. The menu that most often provides these options is the Edit menu.

When users select the menu and see these options, the users know that the application supports the clipboard. These options are not always available, so you must let users know when they can be used. This is accomplished by disabling and graying menu options when they are not selectable. For example, if data is not available, all three items are disabled. When a window contains data that the user can place in the clipboard, the Cut or Copy functions are enabled and the Paste option is disabled. After data has been transferred to the clipboard, the Cut and Copy options should be disabled and the Paste option enabled. Now the user knows when the appropriate functions are available, and the application can provide visual feedback for data selection and placement.

Data selection can be done before the clipboard service is requested or after the service is requested. Preselection functions may be present in another menu option. For example, as a part of an application's data services, an Import menu option or an

Export menu option can be provided. When the Import option is selected and the user presses mouse button one, a tracking rectangle displays, and the user indicates where to place the imported data. At this time, the Paste option should be enabled, and the Copy and Cut options disabled.

When the Export option is selected and the user presses mouse button one, a tracking rectangle should be drawn to enable the user to indicate which data to export. The Copy and Cut options are then enabled and the Paste option disabled.

Postselection functions are provided when the user selects either the Copy or Cut option. While button one on the mouse is pressed, a tracking rectangle is displayed. Once the final selection is indicated, the data is transferred to the clipboard.

The Copy and Cut options are then disabled and the Paste option enabled. Data placement is activated when the user selects Paste. A tracking rectangle is displayed when mouse button one is pressed. After the user indicates where the data is to be placed, the data from the clipboard is transferred to the window.

The use of a tracking rectangle is one means of providing visual cues for data selection or placement. The rectangle is appropriate for graphics applications but is usually inappropriate for text applications.

For text applications, the data to be placed into the clipboard should show its selection by being highlighted. This usually involves redrawing the text in the inverse video of the current screen mode.

Placing data into the clipboard is called *rendering* the data. Rendering may be immediate or delayed. This chapter discusses immediate rendering and then demonstrates how to modify the operation for delayed rendering.

Rendering

When your application performs rendering, a pointer to shared memory is placed in the clipboard. This is done by using the WinSetClipbrdData API. Your application does not have to own the clipboard. The clipboard is owned by the system or an application performing delayed rendering. Once the data has been rendered, the data pointer is invalidated. Your application can no longer address that memory. This is a part of the data integrity the system provides.

Copy

The Copy function is provided so that data can be transferred to the clipboard without destroying the window contents. Once the user has identified the data to transfer, your application will open the clipboard. This prevents other applications from accessing it. The data is copied to a piece of shared memory that has been previously allocated. The data is rendered when WinSetClipbrdData is called. Rendering is specified when

the second parameter to `WinSetClipbrdData` contains the address of the shared memory. After the clipboard is closed, other applications can access the data placed in the clipboard. The following sample demonstrates the instructions used to render data to the clipboard.

```
Allocate_memory();           /* Allocate shared memory */
WinOpenClipbrd(hab);         /* Open the clipboard */
Copy_data();                 /* Copy data to shared memory */
WinSetClipbrdData(hab,      /* Render the data */
                  (ULONG)Mem_addr, /* Shared memory address */
                  CF_DSPTEXT, /* Private text display format */
                  CFI_SELECTOR /* Specify the address format */
                  );
WinCloseClipbrd(hab);        /* Close access to the clipboard */
RemoveRect();                /* Signal Copy complete */
```

Cut

The Cut operation is very similar to copying data into the clipboard. The major difference is that the Copy operation leaves the data intact in the application window, the Cut operation removes it. The procedure for implementing the Cut function is also very similar to the Copy function.

Once the user has identified the data to transfer, the clipboard is opened. This prevents other applications from accessing the clipboard. The data is then copied to a piece of shared memory that has been allocated to hold the data. The data is then removed from the window by erasing that part of the window. The data is rendered when `WinSetClipbrdData` is called, and the second parameter is the address of the shared memory.

Once the data has been rendered, the clipboard is closed. Once that is done, other applications can access the clipboard.

```
Allocate_memory();           /* Allocate shared memory */
WinOpenClipbrd(hab);         /* Open the clipboard */
Copy_data();                 /* Copy data to shared memory */
WinSetClipbrdData(hab,      /* Render the data */
                  (ULONG)Mem_addr,
                  CF_DSPTEXT,
                  CFI_SELECTOR
                  );
WinCloseClipbrd(hab);        /* Close access to the clipboard */
RemoveWindowData();          /* Erase data from window */
RemoveRect();                /* Signal Cut complete */
```

In both the Copy and the Cut operations, the `CFI_SELECTOR` data format is specified as the fourth parameter of the `WinSetClipbrdData` call. This tells the system that the data is a private format. When the data is transferred to the clipboard, it replaces the previous contents of the clipboard.

Paste

When data is transferred from the clipboard to your application, the clipboard must be opened for your program to access data. After the clipboard is opened, your application can access the data and copy the data from the clipboard. After the data has been transferred, the clipboard must be closed. Once this is done, your application no longer has access to the clipboard data. Note that the data is not removed from the clipboard but is simply copied. The data remains in the clipboard until it is overwritten.

```
WinOpenClipbrd(hab);           /* Open the clipboard      */
Mem_addr = WinQueryClipbrdData(hab, /* Get address of clipboard data */
                               CF_DSPTEXT);
Copy_data();                   /* Copy data from the clipboard */
WinCloseClipbrd(hab);         /* Close access to the clipboard */
RemoveRect();                  /* Signal Paste complete      */
```

Delayed Rendering

Applications perform delayed rendering when a number of different data formats are supported by the application rendering the data. The clipboard can have only one copy for each data format at any time, but with delayed rendering, multiple views of the same data may be offered.

Delayed rendering enables you to save time and memory. It also provides some flexibility to the clipboard support. It does require that the source application perform some additional work. The source application must be active before a requesting application can retrieve the clipboard data. Normally the system is the owner of the clipboard, but when delayed rendering is used, the rendering application must own the clipboard. This is done by issuing the `WinSetClipbrdOwner` call.

```
WinSetClipbrdOwner(hab,           /* Set new owner      */
                   hwndClient);
```

In addition to owning the clipboard, your application must handle the `WM_RENDERALLFMTS` message. This message is received when the application owning the clipboard terminates. The formats are then rendered to the new owner. Before termination of your application, you must return the clipboard to the previous owner.

```
WinSetClipbrdOwner(hab,           /* Set no owner      */
                   NULL);
```

An owner of the clipboard must process the `WM_RENDERFMT` message. This specifies where and when the data will be rendered. When the target application calls `WinQueryClipbrdData`, a `WM_RENDERFMT` message is sent to the source appli-

cation. Your application determines whether the requested data format is supported and if so returns the data.

WM_RENDERFMT:

```
if (LOUSHORT(lParam1) == CF_DSPTEXT)
{
    alloc_mem();                /* Allocate memory for data */
    Reformat_data();            /* Format the data */
    WinSetClipbrdData(hab,      /* Render the data */
        (ULONG)Mem_addr,
        CF_DSPTEXT,
        CFI_SELECTOR
    );
}
else
if (LOUSHORT(lParam1) == otherformat)
{
    alloc_mem();                /* Allocate memory for data */
    Reformat_data();            /* Format the data */
    WinSetClipbrdData(hab,      /* Render the data */
        (ULONG)Mem_addr,
        otherformat,
        0L
    );
}
```

Copy

The Copy function remains the same with the exceptions that the second parameter of the WinSetClipbrdData call is set to NULL, and the other formats are set.

```
Allocate_memory();            /* Allocate nonshared memory */
WinOpenClipbrd(hab);          /* Open the clipboard */
Save_data();                  /* Save data */
WinSetClipbrdData(hab,        /* Delayed rendering */
    (ULONG)NULL,
    CF_DSPTEXT,                /* Text format */
    CFI_SELECTOR
);
WinSetClipbrdData(hab,        /* Delayed rendering */
    (ULONG)NULL,
    CF_BITMAP,                 /* Bitmap format */
    0L
);
WinCloseClipbrd(hab);         /* Close access to the clipboard */
RemoveRect();                 /* Signal Copy complete */
```

Cut

The Cut function remains the same with the exceptions that the second parameter of the WinSetClipbrdData call is set to NULL, and the other formats are set.

```

Allocate_memory();          /* Allocate shared memory */
WinOpenClipbrd(hab);        /* Open the clipboard */
Save_data();                /* Save data */
WinSetClipbrdData(hab,      /* Delayed rendering */
    (ULONG)NULL,
    CF_DSPTEXT,             /* text */
    CFI_SELECTOR
);
WinSetClipbrdData(hab,      /* Delayed rendering */
    (ULONG)NULL,
    CF_BITMAP,              /* bitmap */
    0L
);
WinCloseClipbrd(hab);       /* Close access to the clipboard */
RemoveWindowData();         /* Erase data from window */
RemoveRect();               /* Signal Cut complete */

```

Dynamic Data Exchange

Dynamic Data Exchange is the ultimate form of system-provided interprocess communication. It is just like using the telephone. Your application places a call, and one at the other end answers. The DDE process for data transfer is called a *conversation*. After the initial acknowledgment, a conversation takes place. During the conversation, information is passed between applications.

This conversation continues until the application is satisfied with the information received. The same basic elements are present in DDE that are present in a telephone conversation. You may not know the application that you are communicating with, but you will definitely know the topic for data exchange.

If the application that is providing the data is not known at the time the request is initiated, its format is determined before data exchange begins. Each transfer of data is confirmed, so each partner knows the information was received and processed. The data exchange continues until the requesting application decides it is satisfied with the information it has received. The exchange is then terminated by the requesting application. The function is provided through the use of three API calls, WinDdeInitiate, WinDdePostMsg, and WinDdeRespond. DDE can be used only by Presentation Manager applications.

When a dynamic data exchange conversation is initiated, even if the name of the serving application is not known, the request can still be issued. The request will be initiated as a general request using the WinDdeInitiate API.

```

WinDdeInitiate(hwndClient,  /* Initiate a general request */
    "",                    /* Unknown server */
    "NET DATA"             /* Network data statistics */
);

```

The program that services that request gets a `WM_DDE_INITIATE` message and responds with a `WinDdeRespond` call. This call generates a `WM_DDE_INITIATEACK` message. When the requester receives the message, the name of the application that services the request is provided. The next request is made via `WinDdePostMsg`. For example, to request data, you would use the following code:

```
WinDdePostMsg(hwndTo,          /* Handle of server application */
               hwndClient,      /* Your window handle          */
               WM_DDE_REQUEST,  /* Request posted              */
               pDdest,          /* Request packet              */
               FALSE,           /* Do not retry                */
               );
```

The serving application will respond by sending the data, using the `WinDdePostMsg` call.

```
WinDdePostMsg(hwndTo,          /* Handle of server application */
               hwndClient,      /* Your window handle          */
               WM_DDE_DATA,     /* Data posted                 */
               pDdest,          /* Data packet                 */
               FALSE,           /* Do not retry                */
               );
```

You respond with an acknowledgment message to let the server know that you got the data and processed it.

```
WinDdePostMsg(hwndTo,          /* Handle of server application */
               hwndClient,      /* Your window handle          */
               WM_DDE_ACK,      /* Ack posted                  */
               pDdest,          /* Ack packet                  */
               FALSE,           /* Do not retry                */
               );
```

When no more data is needed, the conversation is terminated by sending a terminating message. The other application responds by sending a terminating message.

```
WinDdePostMsg(hwndTo,          /* Handle of server application */
               hwndClient,      /* Your window handle          */
               WM_DDE_TERMINATE, /* Terminate posted            */
               pDdest,          /* Do not retry                */
               FALSE,           /* Do not retry                */
               );
```

The format of the conversation depends on the information your application needs. If your application is going to respond to the `WM_DDE_INITIATE` message, it provides the handle of the window to which the requests are directed. If the program is going to respond to more than one topic, it must use a separate window handle for each topic. The following messages are supported by the dynamic data exchange services.

WM_DDE_ACK	Acknowledges receipt of a message
WM_DDE_ADVISE	Requests an update whenever data changes
WM_DDE_DATA	Notifies requester of availability of data
WM_DDE_EXECUTE	Posted to a server to execute a series of commands
WM_DDE_INITIATE	Requests initiation of a conversation
WM_DDE_INITIATEACK	Gives acknowledgment to start a conversation
WM_DDE_POKE	Requests application to accept unsolicited data
WM_DDE_REQUEST	Requests server to provide data
WM_DDE_TERMINATE	Terminates a DDE conversation
WM_DDE_UNADVISE	Removes an "advise" request

Summary

The data transfer methods discussed are used by Presentation Manager applications only. They ensure data integrity. The clipboard is designed for window applications. The Dynamic Data Exchange does not require the application to have a window, but it must be a Presentation Manager application. An application becomes a Presentation Manager application when it issues a `WinInitialize` call and a `WinCreateMsgQueue` call. The protocol of the DDE now provides a mechanism to confirm the receipt and processing of information.

Subclassing Windows

What Is Subclassing?

When you wish to define the behavior of a window, you write a window procedure for it. This window procedure may then be associated with a class name and attributes using the `WinRegisterClass` function. Windows may then be created using this class.

This provides a great deal of power in how the behavior of different windows is defined. At times, however, you may wish to create a window with most of the characteristics of some other class with only a slight modification. Why should you have to rewrite all the functions of that class to implement it with minor changes?

The answer is that you don't. A feature built into the Presentation Manager enables you to dynamically modify the behavior of a window. This feature is called *subclassing*.

Subclassing a window is the process by which the window procedure for a particular window can be substituted with a different one. This function is available on a per-window basis.

Behind the scenes, a window has a pointer to its window procedure. Where this is stored is of no major concern, only the fact that the pointer is there is important. The OS/2 API provides a function call, `WinSubclassWindow`, to perform this operation. As the name implies, this causes a window to be subclassed.

Performing the Subclass

The subclassing procedure can replace one window procedure with another. Simply, it takes the pointer to the window procedure that controls the window's behavior and replaces it with a pointer to another window procedure. The call to `WinSubclassWindow` looks like this:

```
PFNWP OldWindowProc;  
  
OldWindowProc = WinSubclassWindow(hwndMyWindow,  
                                   (PFNWP)NewWndProc);
```

When your program makes a call to `WinSubclassWindow`, you supply the handle of the window to be subclassed and the name of the new window procedure. The return value is a pointer to the previous window procedure.

Once this code is executed, all messages go to the new window procedure, whether they are sent (a direct call to the window procedure) or are posted to the queue and then dispatched. This new window procedure acts as the window's controller. The old window procedure is returned by `WinSubclassWindow` to provide a mechanism by which the window's prior functions may be called when necessary.

Unsubclassing

There may be instances when you wish to subclass a window for only a short period of time. It is then necessary to undo the subclass. To do this, subclass the window again using the original window procedure for the "new" procedure. An example of this is

```
/* This code will undo the subclass that was executed before */  
  
OtherProc = WinSubclassWindow(hwndMyWindow,  
                               (PFNWP)OldWindowProc);
```

Once this code executes, the original window procedure becomes the recipient of a message sent to the window. The subclass procedure is no longer in the call or return chain.

Using the Subclass Procedure

A window procedure that is specified in a `WinSubclassWindow` call is no different than any other window procedure. The key to effective subclassing is to use the "old"

window procedure as you would `WinDefWindowProc`. This is often referred to as the “return chain.”

The return chain is the path that execution takes when returning from different levels of function calls. For example, when a window procedure is called, execution is begun. Say this message is not important to the window. A call is made to `WinDefWindowProc` to handle it. When this call is made, another level has been “nested” in the call hierarchy.

When `WinDefWindowProc` is finished processing, it executes a return. This return brings execution back to the window procedure. A return is then executed, which gives control back to whoever made the call to your window. This return chain works back up to the originator. You must be aware of this when subclassing. When done properly, manipulating the return chain can be very effective in reducing the amount of work your subclass procedures must do.

A procedure you change to by subclassing should only handle those tasks that differ from the default behavior of the previous window procedure. The reason behind subclassing is to take a previously defined class, whether it be application- or system-defined, and use it, but with a few modifications. The goal is to keep the code in the subclass procedure to a minimum.

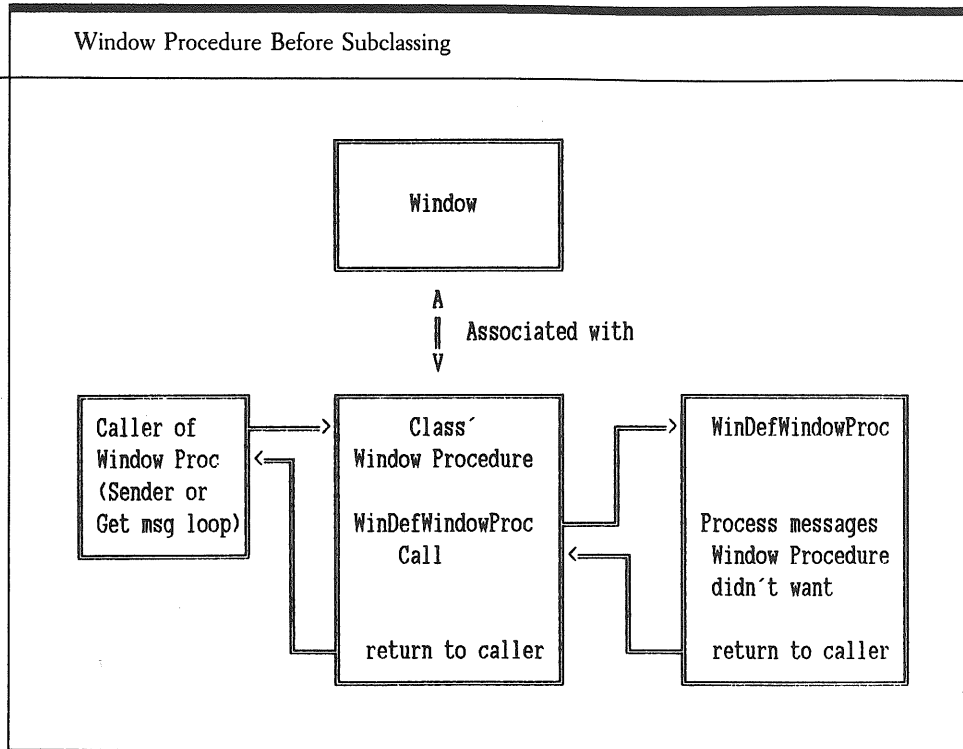
The process used to keep the code short and to the point follows the same basic principles as any other window procedure: Look for what you are interested in and pass the rest to the default procedure. The default procedure in this case is the old window procedure that was returned from `WinSubclassWindow`. Another level in the return chain is added when this is done.

In the subclass procedure, all the functions that modify the behavior of the original procedure are coded. In the discussion of messages, you have seen that each message's return value indicates to the caller whether the caller should take any action on the message. It is crucial in subclassing that return values be kept consistent. The default behavior of a window is being modified, and care must be taken to indicate to the original window procedure what has been done. The diagram in Figure 23.1 shows the normal chain of events before a window is subclassed and `WinDefWindowProc` is used to execute the “default” processing.

In Figure 23.2, you see how the subclass procedure is called in place of the original window procedure. To keep the code in the subclass procedure isolated to only those items that are changed from the default, only the differing code is executed, and a call to the old window procedure is made for everything that remains the same. This window procedure handles all the functions that are not modified. As usual, it handles any of the ones it is interested in and calls `WinDefWindowProc` for those it does not wish to process.

A layer to the call chain for messages has been added. You may ask why you should do this. Why can't you just code for the message inside the window procedure? The answer is that in many cases, you have not coded the window procedure for the window.

Figure 23.1



For example, you may want to force all input going into an entry field you have created to be uppercase. Listing 23.1 shows an example of how to do this.

Listing 23.1

```
PFNWP OrigProc;
```

```

/*****
/*
/* This function call will subclass the window to place
/* the procedure first in the call and return chain.
/*
/*
/* OrigProc is the original entry field window procedure.
/* hwndEntryField is the entry field's handle.
/* UpperCaseProc is the window procedure subclassed to.
/*
*****/

```

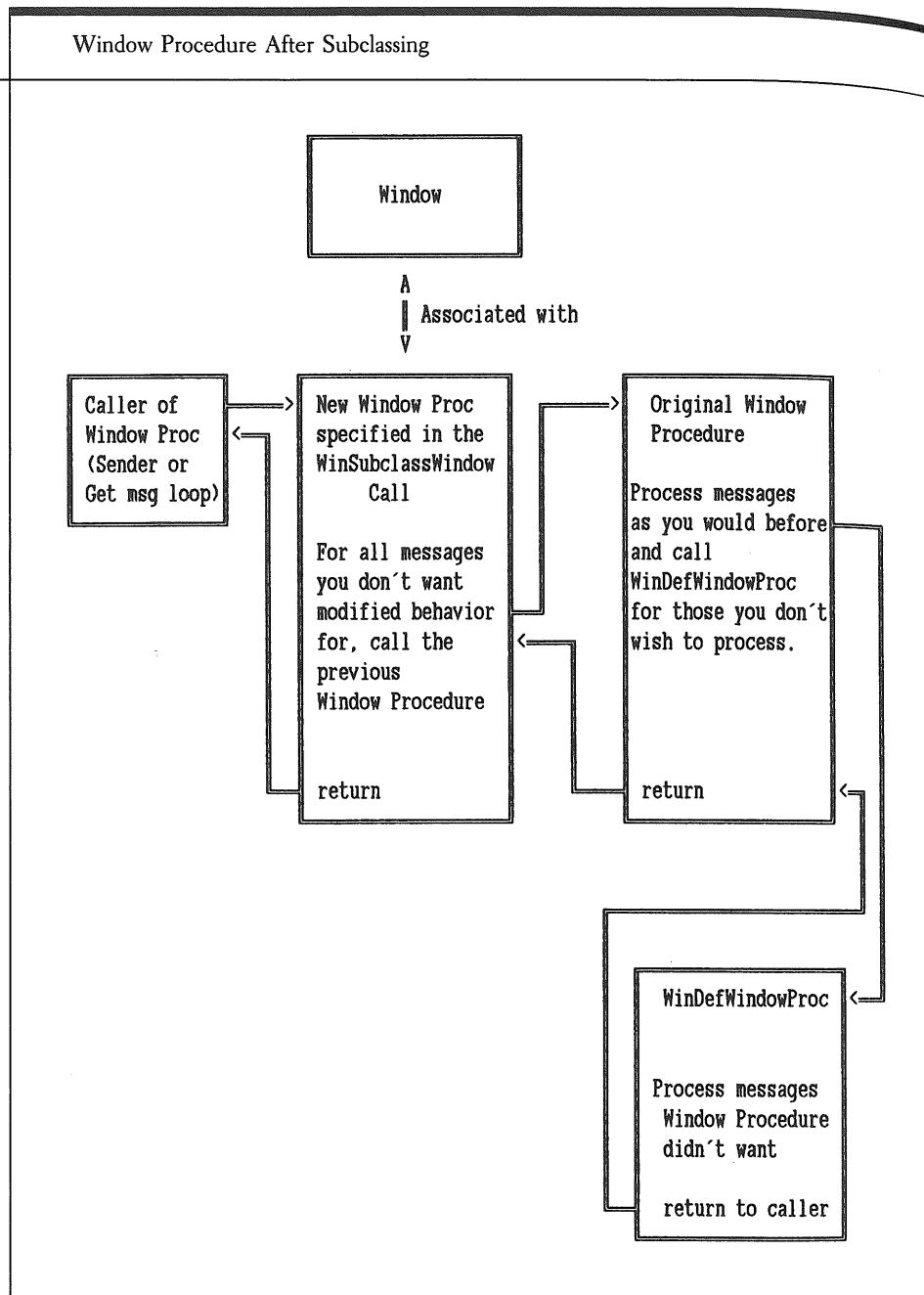
```

OrigProc = WinSubclassWindow(hwndEntryField,
                             (PFNWP)UpperCaseWP);

```

Figure 23.2

Window Procedure After Subclassing



```

/*****
/*
/* This is the subclass procedure to handle uppercasing entry fields.
/* It modifies the character code only if it is lowercase. It does
/* not process the message. The entry field procedure should do that.
/* This is only modifying the keystroke message; it lets the entry field
/* do the rest.
/*
/*
*****/

MRESULT EXPENTRY UpperCaseWP(HWND hwnd,USHORT message,MPARAM mp1,MPARAM mp2)
{
PCHARMSG p;                /* Pointer to access the entire character message */
                           /* This will be used with helper macro CHARMSG */

    switch(message)
    {
        case WM_CHAR:                /* For the WM_CHAR message */
            p=CHARMSG(&message);      /* get the char message */
            if((p->fs & (KC_KEYUP|KC_CHAR))==KC_CHAR) /* You only want
            {                               /* downstroke messages */
                if(isalpha(p->chr) && islower(p->chr)) /* If keystroke was lower-
                p->chr=(USHORT)_toupper((UCHAR)p->chr); /* case, make it upper
            }

        /* Now, return this message to the original window procedure. You
        /* modified it but did not process it completely. You now wish the entry
        /* field procedure to handle it as any other. All this did was modify
        /* the keystroke.

        *****/

        return((*OrigProc)( hwnd, message, mp1, mp2 ));
    }
}

```

In Listing 23.1, you see how an entry field may be subclassed to force all its input to be uppercase. In this case, you are not handling the message in the subclass procedure. The keystroke that the entry field's default window procedure sees is simply modified. In essence, this adds a type of monitor to this window's processing.

Because the entry field does not provide this function in its own window procedure, you need to subclass. This could be coded into your own window procedure, but then you would have to also code all the functions the entry field default window procedure provides. This provides a much more efficient solution.

Watch That Return Chain!

Subclassing provides substantial freedom to modify the behavior of instances of previously defined classes. Be careful of how you use this power, however. When a window

is subclassed, a major modification is made to its call chain of window procedures. Care must be taken to ensure that the return chain finishes where it started.

There are many execution paths a subclass procedure may take. In the previous example, the code simply modifies a keystroke and sends it to the original procedure. There may be cases when you wish to take over processing of a message completely and not allow it to reach the original window procedure or `WinDefWindowProc`. There are several techniques you can use in subclassing windows and controlling their return chains.

The first case is when you wish to allow the message to continue to the original window procedure (which is usually the procedure associated with the class). As in the previous example, a call is made to the original procedure under all circumstances. Whatever is done in this procedure, the prior window procedure is called before returning. In this case, the return value from the subclass procedure is simply the return value from the call to the prior procedure.

Another case is when you wish to prevent a message from reaching a window procedure that might do some processing as a result of it. Say, for example, there is a message you wish to stop before it gets to the original window procedure or `WinDefWindowProc`. In this case, you can simply return from the subclass procedure by returning the value for the message that indicates that no action has been taken, but the message has been processed.

In this case, remember to only look for the message you wish to stop and pass all others to the original window procedure as if this new procedure were not there. In this case, the procedure is acting as a message filter.

A third case is when you wish to process a message in your subclass procedure rather than allowing the default procedure for the class to handle it. In this case, the procedure should look for that message and process it as it would in any other window procedure. The proper value can be returned to the caller without the original procedure or `WinDefWindowProc` ever seeing it. It is important in this case as well to make a call to the original procedure for all other messages.

These are not all the instances for which you might wish to subclass a window, but they should provide an idea of the responsibility of a subclass procedure.

Multiple Subclasses

The final concept to understand about subclassing is multiple subclasses. A window may be subclassed as many times as desired. Each subclass procedure must follow the rules of the return chain. Many times programmers code to function names directly. This is sometimes called “hardcoding” names into a function. Because the call and return chain of a window may be dynamically changed by subclassing, take great care when you use multiple subclasses.

A good convention for working with multiple subclasses is to only subclass the window procedure that is the first in the call chain and only unsubclass the one that is first. The subclass "chain" should be treated like a last-in, first-out queue. Only subclass the top of the queue and only remove a subclass procedure from the top. This ensures consistency in your call and return chain.

Summary

Subclassing is a powerful tool. It provides a way to modify the behavior of windows that have already been defined and created. It is especially useful in altering the behavior of instances of the system-defined classes such as a frame or entry field window.

Subclassing simply alters the call and return chain between window procedures. Once a window is subclassed, messages go to the new procedure until such time as the window is destroyed or the unsubclass is performed. With this feature you can tailor any window to behave as you wish without recoding all the functions the window presently has.

Placing Resources in Dynamic Link Libraries

You have seen that a dynamic link library (DLL) is a means of sharing code and/or data. When you are building a large application, routines that are used many different places are candidates for a dynamic link library. The entire application may even be a dynamic link library if it provides services that are used by other applications. This discussion probably leads you to believe that only executable code can be placed in a dynamic link library, but that is not the case. In the Presentation Manager system, dynamic link libraries can be used to hold fonts and application resources. The benefits of using DLLs are that the application's size is reduced and the library contents can be shared by other applications. When fonts are placed in a dynamic link library, the library file is called a *font library*.

When resources such as icons or bitmaps are placed in a dynamic link library, the library file is called a *resource library*. The two cannot be mixed, because they are loaded and their contents retrieved by different methods, which are discussed shortly. The one feature that they have in common is that each must have a "stub" dynamic link library.

A stub DLL is a file that has been formatted with the appropriate header information for a dynamic link library. A stub has no code or data.

Creating a Dynamic Link Library Stub

The contents of a stub will not be discussed in depth here, because you will never have to reference any part of the stub information. The system manages and maintains

stub information. What is important to know is how the stub is created. The names used are generic names that you can change. The first step in creating a resource DLL is to create the source file. Sample code to create it is

```
int ARCTUSED = 1;
void far pascal stub_name()
{
}
```

The first statement tells the C compiler that you are creating a DLL. The second statement assigns a name to the function. It defines the function as a "far" function that uses the "pascal" calling conventions.

This function accepts no data and returns no data. The third statement indicates the start of the function. The fourth statement indicates the end of the function.

As you can see, the function involves no executable statements. The file is saved with a name of STUB.C. The next step is to create the object file STUB.OBJ. You create STUB.OBJ by compiling STUB.C. The following command line is used to create STUB.OBJ:

```
CL -c -Gs stub.c
```

The -c parameter tells the compiler to stop after the .OBJ file is created. The -Gs option tells the compiler not to generate stack-checking code. The result of the compile is STUB.OBJ. The object file is next used as input to the linker to create the stub dynamic link file STUB.DLL. Before linking the stub, you must create the other input file that the linker needs to create the dynamic link file STUB.DLL. That other input file is the module definition file, STUB.DEF. A sample STUB.DEF is

```
LIBRARY stub_name
DESCRIPTION 'DLL stub file'
DATA NONE
```

This code has the minimum set of statements required in the module definition file for a stub dynamic link file. The first statement tells the linker that the output is a DLL. The second statement contains a comment that is placed in the DLL stub. The third statement tells the linker that there are no default data attributes to be defined for this library. The command line used to create the dynamic link library stub is

```
Link STUB.OBJ, STUB.DLL, nul, /NOD, STUB.DEF;
```

The linker links the STUB.OBJ file, using the directives from the STUB.DEF file to create the STUB.DLL file. The nul parameter tells the linker not to generate a .MAP file. The /NOD parameter tells the linker not to use the default libraries, such as OS2.LIB. The output from this operation is the stub DLL. This is the file that the font (or resource) will be added to if you are creating a font library. If you are creating a resource library, the resources are added to this file.

Creating a Font Library

The information explained here enables you to create your own character fonts or pattern fonts. All fonts within a single font library always belong to the same group or family. This example describes how to use the Font Editor to create a cursive font.

The first step is to repeat the procedure used to create the stub DLL. Simply replace all occurrences of *STUB* or *stub_name* with *CURSIVE*. For example, *STUB.C* becomes *CURSIVE.C* and *stub_name()* becomes *cursive()*. The next step in the process is to create the .FNT file. The Font Editor is used to create this file.

The Font Editor is one of the tools provided in the OS/2 Programmer's Toolkit. The Font Editor is used to create image fonts that are either fixed spaced or proportionally spaced. It also creates pattern fonts.

When the Font Editor is started, a copy of the system default font is loaded. The system font is a proportionally spaced image font. The system font has a font name (or face name) of System Proportional. This must be changed to Cursive.

Start the Font Editor and select the Header menu item. When the menu appears, select the Naming... option. You must change the face name from System Proportional to Cursive, then press Enter. The system font contains the code page's 850 character images for the ASCII values 0 through 255. If you change from a proportionally spaced to a fixed spaced font, you must redefine all the character images.

If you are creating a pattern font, the size is set to 8×8 . Because the cursive font you are creating is proportionally spaced, you have to modify only the character images, A-Z and a-z. You next change each character in the font. The characters are changed so they appear joined when placed next to each other. Columns are added to the left or right as necessary, using the Width menu item. After you have modified the characters, save the new font. The font is now saved in a file named *CURSIVE.FNT*.

Next, you must create a resource file called *CURSIVE.RC*. The *CURSIVE.RC* file is a text file created with a text editor. The *CURSIVE.RC* file contains the following line:

```
rcinclude CURSIVE.FNT
```

The resource file *CURSIVE.RC* is then compiled with the resource compiler. The result of this is the binary output *CURSIVE.RES*. This is the information that is placed in the dynamic link stub, *CURSIVE.DLL*. The command line used to add this to the DLL is

```
RC CURSIVE.RC CURSIVE.DLL
```

The final step in the process is to rename the file to *CURSIVE.FON*. This file is then placed in a directory that is pointed to by the *LIBPATH* statement of the *CONFIG.SYS* file. This is the file that is referenced in the *GpiLoadFont* call.

Creating a Resource Library

The first step in creating a resource library is to create a stub DLL. You will change the name to that of your application, for example, STUB to MYAPP and stub_name() to myapp().

A resource library is a file that contains a number of resources used by your application. You use a text editor to create the resource file. The contents of the resource file are all the menus, icons, string tables, and other Presentation Manager resources used by your applications. The resource file is compiled and the resulting .RES file is placed in the dynamic link library. The following statement shows the command line used to create the resource library:

```
RC MYAPP.RC MYAPP.DLL
```

The final step in the process is to place the resource library in a directory referenced by the LIBPATH statement in the CONFIG.SYS file. Your application must be modified to access this resource library. The resource library is loaded as a part of your application's initialization. The statement used to load the resource library is DosLoadModule. This call loads the module and returns a handle. This handle is later used to reference the resources in the DLL. The DosLoadModule call is coded

```
rc = DosLoadModule(NULL,          /* Error buffer          */
                   0,             /* Error buffer length   */
                   "MYAPP",       /* Name of module to be loaded */
                   &rhandle,      /* Returned module handle */
                   );
```

If the function fails, an error return code is placed in the variable rc. The error buffer is ignored for this example.

There are certain Presentation Manager function calls that are specifically allowed to load resources from DLLs. Following is a list of them:

- GpiLoadBitMap
- WinLoadAcceleratorTable
- WinLoadDlg
- WinDlgBox
- WinLoadMenu
- WinLoadPointer
- WinLoadString
- WinCreateStdWindow

WinCreateFrameControls

WinCreateWindow

Summary

As you have seen, the process of creating a stub dynamic link library is very simple. The only change required from any other DLL is the change of the name used. Fonts and resources cannot be placed in the same library, because they are loaded by different means. Both methods are similar: They both require the use of the resource compiler to place the information in the dynamic link stub.

Once the resources or fonts are in the DLL, they may be accessed by any number of programs simultaneously. An entire product may be a set of fonts that can be loaded by a program. All the OS/2 resources, such as icons and pointers, reside in DLLs provided with the system. This allows many programs to use the resource without taking up storage for duplicate copies.

Application Debugging Tips

As talented programmers all know, no one is protected from bugs that can crawl around inside programs. The Presentation Manager interface offers new challenges in debugging programs.

When you sit down to debug a program, think about the type of program you have written and the environment that program runs in. These items determine which debugging methods are available to you.

There are two ways to approach the search for and correction of program errors. You can debug with or without the aid of a debugging tool. This chapter analyzes the different program types and the environments in which they run. The ways this complex environment can be simplified and debugging made easier are explored. In addition, the tool appropriate for debugging each type of program and the program preparation necessary to aid debugging efforts are discussed.

There are three types of programs you can write: device drivers, dynamic link libraries, and user (application) programs. These are considered to be three different types because each has a different structure and interfaces differently to the system.

Device drivers and dynamic link libraries (DLLs) are like subroutines. Device drivers are called by the kernel, and DLLs are called by an executing program or the operating system. Device drivers are started at system initialization time with a `DEVICE=driver_name.SYS` statement in the `CONFIG.SYS` file. DLLs are executed when a program makes a call to the library, and a user program is started directly by the user or from a batch file.

Programs can be written to perform a single activity or multiple, concurrent activities. The single-activity program is called a *single-tasking program*. The multiple, concurrent

activity program is called a *multitasking program*. Multitasking is achieved through the use of multiple threads, multiple processes, or multiple sessions. Programs running in the multitasking environment are susceptible to possible timing or multiple-access errors in addition to logic errors.

Timing errors occur when two or more threads or processes are running concurrently and one provides input for the other. Multiple-access errors occur when two or more threads have access to common data areas. They can “bump” into each other accessing the same items at the same time. This is a classic problem in multitasking programs.

The easiest type of program to debug is a single-tasking program. In this environment you only have logic errors to solve. That leaves the job of simplifying the multitasking environment. Multiple processes and multiple sessions may be treated the same because either way, another process is being run.

The parent process should be tested first to ensure that it is working properly. The actions of the child process can be simulated where necessary. Once the parent process is debugged, you can turn your attention to the child process.

If different processes run asynchronously and depend on each other for input, you should debug first for multiple access errors and then work on any timing errors. A similar approach is applied to multithreaded programs.

You should debug the first thread first, before you run the other threads. Then, debug each of the other threads one at a time. Once the threads are debugged separately, your attention can be turned to asynchronous thread execution in the same way as processes.

Free Style Debugging

Program errors can be found and corrected without the aid of a debugger. This approach is referred to as “free style” debugging. In an OS/2 full-screen session, C `printf` statements are frequently used to print the contents of program variables and state during execution. A similar method is to write these statements to a disk file so they can be examined later.

This approach can be used to find timing and multiple-access errors. This method is appropriate only for user programs and DLLs. Presentation Manager programs cannot use the `printf` approach. The `printf` function is a text-only function and cannot display text on a graphics screen. However, a facility offered by the Presentation Manager interface provides a similar function.

The Presentation Manager session offers message boxes to inform users of events occurring within the program. They may also be used as a debugging aid.

The following `printf` function provides an example of how to display the contents of the variable `count`.

```
printf("This is contents of count = %d ",count);
```

You can get the same results in a Presentation Manager program using the following instruction sequence.

```
char work_buf[80];
.
.
/* Copy string into a character array */
sprintf(work_buf,"This is contents of count = %d ",count);
WinMessageBox(HWND_DESKTOP, /* Owner of display window */
              HWND_DESKTOP, /* Parent of display window */
              work_buf, /* Address of string to display */
              "Debug Message Display", /* Window title */
              1, /* Display window ID */
              MB_OK /* Pushbutton used to dismiss window */
              );
```

This example places a formatted text string into a character array. WinMessageBox is used to display the text in a dialog window. The dialog window has the title Debug Message Display. The displayed text is shown in the window with the OK pushbutton below the text. Once you have seen the message, you can press Enter to dismiss the window and allow the program to continue.

You should select a few key places to display debugging information. If you have a large amount of information to be displayed, use a debug output file. If there is not much to display, a message box is quite sufficient. These debugging statements must be inserted into the source code of the program. The process is an iterative one and is time consuming.

The file must be edited to insert the debugging statements, and the program must be rebuilt. You repeat the process until the error is found.

Hardware Debugging

There are two types of debugging tools generally available. You can use either a symbolic debugger or a hardware debugger to find and correct problems. These methods of debugging require some preparation before you can start troubleshooting errors. First, a hardware debugger and the requirements to use it are discussed; then we explore a symbolic debugger.

The hardware debugger most often used is an "in circuit emulator." This type of debugger hooks in directly with the computer hardware and ties into the processor. This type of debugger can be used to debug all three types of programs—device drivers, DLLs, and user programs.

When debugging a program, you need to insert a *breakpoint* at the places you want the program to stop, enabling you to look into it. A breakpoint is simply a point in the

program that you specify to halt execution. When using a hardware debugger, you are required to establish the first breakpoint and then you can use the machine to set others. You can also establish other predefined breakpoints in your program. To code a breakpoint for the hardware debugger, place an INT 3 instruction at the desired point in the program.

This is relatively easy for device drivers, because they are usually coded in assembly language. In a DLL or user program (which is usually written in C), you can call an assembly language routine that issues the INT 3 instruction.

What happens if you do not have access to an assembler or you do not know how to write an assembly language program? You can use the following technique to accomplish the same results.

```
char far INT3D[8]={0x55,      /* PUSH BP          */
                  0x8B,0xEC, /* MOV SP,BP        */
                  0xCC,      /* INT 3             */
                  0x8B,0xE5, /* MOV BP,SP        */
                  0x5D,      /* POP BP            */
                  0xCB};     /* Far return        */
int (far *N3)(void);         /* Pointer to a function */

                                /* Get Code selector for INT 3 */
                                /* instruction array           */
DosCreateCSAlias((SEL)HIUSHORT(int3d),
                (PSEL)&(SELECTOROF(*int3)));
.
.
.
(*int3());                  /* Call the INT 3 routine */
```

Using the feature of OS/2 that allows programs to access pieces of data as executable code (DosCreateCSAlias), the INT 3 routine can be defined as a piece of data, INT3D, and executed as code.

A pointer to a function is defined. As a part of the program initialization, you can obtain a code selector for the INT 3 routine defined in the data segment so that the five instructions that perform the INT 3 instruction can be executed. The selector of the INT3D data segment is passed to the function. The system then returns a code segment "alias" in the function pointer selector. This INT 3 routine may then be used throughout the program.

Once you have established the breakpoints, you compile your program or DLL with the -Fc option. This creates a .COD listing file that comprises a mixture of C source statements and the generated assembly language instructions for each source statement. When you link the program, be sure you specify the /M option so you will get an extended map file. The extended map file and the mixed source listing make it easier to use the hardware debugger by providing cross references between the assembly language instructions and the C source code.

Once you take these steps, you can debug the high-level language program with the hardware debugger.

Symbolic Debugging

The other tool used to debug programs is a symbolic debugger. There are a number of symbolic debuggers available, but the CodeView™ debugger will be used in this discussion.

The CodeView debugger can be used to debug DLLs as well as user programs. The CodeView debugger should be started in a full-screen session when you are debugging a Presentation Manager program. A C program you debug with the CodeView tool must be compiled with the `-Zi` and `-Od` flags.

The `-Zi` option produces CodeView information, and the `-Od` option turns off compiler optimization. When you link your program, the `/CO` option must be used. This makes the symbolic information a part of the executable module. First, symbolic debugging for a DLL is discussed. We then move on to symbolic debugging of user programs.

Once the DLL is built, you need a test program that calls the routines that are being debugged. The CodeView debugger is started to debug the test program. As a part of the parameters, specify the `/L` option and give the name of your DLL (called `TEST`). The statement used is similar to

```
CVP /L TEST.DLL TESTPGM.EXE
```

You are able to “single step” through your DLL just as you do with a user program. If you have to remove symbolic information from some parts of your library, the `.COD` mixed listing file and the map file help you keep track of where you are.

The preparation for debugging user programs is similar to that required for debugging DLLs. The differences are that the user program can be started and tested directly, and the CodeView debugger is started differently. If you are testing a single-tasking program called `TEST`, you start it by issuing the following command from the system prompt:

```
CVP TEST
```

If you are debugging a program that requires the use of the mouse, your debug session must be started with the mouse disabled; otherwise, the CodeView debugger will use it. This is done using the following command:

```
CVP /M TEST
```

By entering this you can put a breakpoint in your program and then start program execution. When the mouse button is pressed and your breakpoint is executed, you

can press the button while you are in the CodeView session so that the results are not affected when you return to the Presentation Manager session. This is most useful when you are debugging window scroll bars and mouse interaction with menu selections.

Program preparation remains the same when you are debugging multithreaded programs. The start of the debug session remains the same as that of the single-tasking program. You will use the CodeView "thread commands" to debug and control program threads. You can set breakpoints in multiple threads at the same time. This helps when you are debugging asynchronously executing threads.

The last multitasking scenario is the multiprocess program. The programs are prepared the same way the single-tasking program is prepared. The differences are how the CodeView debugger is started and how the multiprocess commands are used.

After you have separately debugged the parent process and child process, start the CodeView debugger and use the /O option to indicate that any child processes will be debugged as an extension of the parent. The command you enter from the command line prompt is

```
CVP /O TEST
```

When the child program is started, the CodeView debugger asks you if you want to debug the child process. If you reply yes, the CodeView debugger starts another copy of itself, and the child process. This starts another independent debugging session for the child process.

Summary

You have seen how you can debug your program with and without a debugger. There is nothing that prevents you from using both at the same time. Quite often it is advantageous to combine both the free style method and use of a debugger when you troubleshoot a program.

Everyone has individual debugging preferences, but the Presentation Manager session presents special considerations. This chapter's discussion of free style debugging and the debugger are just two of the possible techniques.

Advanced VIO Programming

This chapter explains what makes a program an Advanced Video Input/Output (AVIO) program. Then a sample program is dissected to see what makes it different from a “normal” windowed program.

An AVIO program is a text-only program. An AVIO program is a Presentation Manager program that uses the window management features of the Presentation Manager system along with a subset of the base video API functions to display text. This is the key difference between an AVIO program and a “normal” Presentation Manager program. The normal Presentation Manager program uses the graphics interfaces to display text in its windows and the AVIO program uses only the base video calls. An AVIO program’s text is displayed in fixed columns and fixed rows.

The Presentation Manager AVIO support provides three attributes per character, two more than the base video supports. When an AVIO program is created, an AVIO presentation space must be created and used with the base video calls. In the base video APIs, this parameter is called the *video handle*. In full-screen programs, this is specified as zero. In an AVIO program, it is the AVIO presentation space handle. An application using these extended attributes can use 16 foreground and 16 background colors instead of the 8 colors supported by the base. The underline and the reverse video attributes are supported in the second attribute. The third attribute is for private application use and is not used by the system. This attribute can be used for anything you might like.

Although you can change the font in an AVIO program, the characters that actually print depend on the capability of the user’s output device. The EGA and VGA displays support two character sizes, 8×8 and 8×12 for the EGA, whereas the VGA display

supports 8×8 and 8×14 . The 8514/A display supports 7×16 and 12×20 sizes. You can switch between these sizes by changing the device cell size.

To change the font type, the `VioCreateLogFont` function must be used. This is not demonstrated here, because it requires the use of a fixed spaced font that has one of the sizes just specified, and the other fonts are all proportionally spaced.

Listing 26.1 is a sample AVIO program that demonstrates some of the attributes just discussed.

Listing 26.1

```

/*****
/*
/* Program name: AVIOPGM.C
/*
/* Description: This program displays 36 lines of text. Each line
/*               contains 132 characters. The first 80 characters are in
/*               lowercase, and the last 52 are in uppercase. You can
/*               change the base attribute colors. The foreground colors
/*               are changed by selecting an item from the first menu.
/*               The background color is changed by selecting an item
/*               from the second menu. You can select or unselect the
/*               underline and the reverse video attributes from the
/*               extended attribute. To see the effect of the change,
/*               simply use the cursor movement keys, the PgUp or PgDn
/*               keys, or the horizontal/vertical scroll bar with
/*               the mouse.
/*
/* Required Files:
/*               AVIOPGM.C   Source program
/*               AVIOPGM.H   Header file
/*               AVIOPGM.RC  Resource file
/*               AVIOPGM.DEF Module definition file
/*               AVIOPGM.MAK Make file used to build program
/*
*****/

/***** Include relevant PM headers *****/

#define INCL_WINERRORS
#define INCL_GPI
#define INCL_WINSYS
#define INCL_WINFRAMEGR
#define INCL_WININPUT
#define INCL_WINMENUS
#define INCL_WINSCROLLBARS
#define INCL_AVIO
#define INCL_BASE

/***** Include files *****/

#include <stdio.h>

```

```

#include <ctype.h>
#include <string.h>
#include <os2.h>
#include "aviopgm.h"

/***** Global data *****/

HAB hab;
HWND hwndFrame;
HWND hwndClient;
HDC hwndClientDC;
HVPS hvps;
struct {
    CHAR chr;
    BYTE baseattr;
    BYTE extattr;
    BYTE resvattr;
} chcell;
FATTRS viofattr;
char aviostr[36]={'a','b','c','d','e','f','g','h','i',
                'j','k','l','m','n','o','p','q','r',
                's','t','u','v','w','x','y','z','1',
                '2','3','4','5','6','7','8','9','0'};
SHORT rows, columns, old_columns, old_rows;
SHORT hslrpos;
SHORT vsldrpos;
SHORT cursor_col, cell_width, vis_col;
SHORT cursor_row, cell_height, vis_row;
MRESULT FAR PASCAL AvioWndProc(HWND hWnd, USHORT msg, MPARAM mp1, MPARAM mp2 );

/***** Start of main procedure *****/

cdecl main()
{
    HMQ hMsgQ; /* Message queue handle */
    QMSG qMsg; /* Queue message structure */
    ULONG ctldata; /* Frame control data */

    hab = WinInitialize(NULL); /* Initialize PM access */
    hMsgQ = WinCreateMsgQueue( hab, 0 ); /* Create message queue */

    WinRegisterClass(
        hab, /* Anchor block handle */
        "AVIOWIN", /* Window Class name */
        AvioWndProc, /* Address of window proc. */
        (ULONG)CS_SIZEREDRAW, /* Class style */
        0 /* No extra window words */
    );

    /* Standard window without
    /* an accelerator table and
    /* an icon.

```

```

ctldata = FCF_STANDARD & FCF_ACCELTABLE & FCF_ICON;

/* Add vertical scroll bar */
/* and horizontal scroll */
/* bar to the standard */
/* window controls */
ctldata = ctldata | FCF_VERTSCROLL | FCF_HORZSCROLL;
hwndFrame = WinCreateStdWindow(
    (HWND)HWND_DESKTOP, /* Desktop window is parent */
    (ULONG)WS_VISIBLE, /* Window style */
    (PULONG)&ctldata, /* Frame control data */
    "AVIOWIN", /* Window client class */
    "Sample AVIO Window", /* Title bar string */
    WS_VISIBLE, /* Window style - visible */
    NULL, /* No module handle */
    1, /* Window ID */
    (HWND)&hwndClient /* Client window handle */
);

/* Main loop */
while( WinGetMsg( hab, (PQMSG)&qMsg, (HWND)NULL, 0, 0 ) )
{
    WinDispatchMsg( hab, (PQMSG)&qMsg );
}
VioAssociate(NULL,hvps); /* Disassociate PS */
VioDestroyPS(hvps); /* Destroy PS */
WinDestroyWindow( hwndFrame ); /* Destroy the window */
WinDestroyMsgQueue( hMsgQ ); /* Destroy the message queue */
WinTerminate( hab ); /* terminate PM access */
}

/***** End of main procedure *****/

USHORT lastfitem = 101;
USHORT lastbitem = 201;
PERRINFO perrinfo;

/***** Start of Window procedure *****/

MRESULT FAR PASCAL AvioWndProc( HWND hWnd, USHORT msg, MPARAM mp1, MPARAM mp2 )
{
    RECTL rect;
    BYTE back_clr, fore_clr;
    HWND wnhnd;
    HPS hps;
    USHORT attrstate;
    USHORT buf1 = 1;
    USHORT rc;

    switch( msg )
    {

```

```
case WM_CREATE:
```

```

                                /* Get client window DC      */
    hwndClientDC = WinOpenWindowDC(hwnd);

    VioCreatePS((PHVPS)&hvps,    /* AVIO PS handle      */
               36,              /* PS is 255 bytes long */
               132,             /* PS is 132 bytes wide */
               0,               /* PS format            */
               3,               /* Three attributes per char */
               0);             /* Reserved             */

```

```

    WinSetWindowPos( hwnd,      /* Shows and activates frame */
                    HWND_TOP,   /* window at position 00,    */
                    00, 50, 500, 300, /* 50, and size 500, 300    */
                    SWP_SIZE | SWP_MOVE |
                    SWP_ZORDER |
                    SWP_ACTIVATE | SWP_SHOW
                );

```

```

    VioAssociate(hwndClientDC, hvps); /* Associate PS with DC      */

    chcell.baseattr = CLR_NEUTRAL;    /* Initialize base attr      */
    chcell.extattr = 0;               /* Initialize extended attr  */
    chcell.resvattr = 0;              /* This attr is always 0    */

```

```

                                /* Write canned data to the */
                                /* VIO PS. 36 lines each   */
                                /* 132 bytes                */

```

```

    for (cursor_row = 0, cursor_col = 0;
        cursor_row < 36; cursor_col++, cursor_row++)
    {
        chcell.chr = aviostr[cursor_col]; /* Assign letter from alphabet */
    }

```

```

/*****
/* Write 80 bytes of lowercase data
*****/

```

```

    VioWrtNCell((PBYTE)&chcell, /* Cell to be written      */
               80,              /* Repeat count            */
               cursor_row,      /* Starting row position for output */
               0,               /* Starting col position for output */
               hvps );          /* Video handle            */
                                /* Convert letter to uppercase */
    chcell.chr = toupper (aviostr[cursor_col]);

```

```

/*****
/* Write 52 bytes of uppercase data
*****/

```

```

                                /* Cell to be written */
VioWrtNCell((PBYTE)&hcell,
            52,                /* Repeat count */
            cursor_row,        /* Starting row position for output */
            80,                /* Starting col position for output */
            hvps );            /* Video handle */
}
cursor_row = cursor_col = 0; /* Initialize cursor variables to 0 */
VioSetCurPos(cursor_row, cursor_col, hvps); /* Set cursor position */
                                /* Initialize variables used to track */
vis_row = vis_col = 0;        /* upper left corner of window */
break;
case WM_COMMAND:

/*****
/* PM sends a WM_COMMAND message when the user selects a menu item */
*****/

                                /* If the first menu command */
                                /* is selected */
if (((SHORT1FROMMP(mp1)) > IID_FOREGROUND) &&
    ((SHORT1FROMMP(mp1)) < IID_BACKGROUND))
{
                                /* Get menu handle */
    wnhnd = WinWindowFromID(hwndFrame, FID_MENU);
                                /* Uncheck the previously */
                                /* selected menu item */

    WinSendMsg(wnhnd,MM_SETITEMATTR,
                (MPARAM)MPFROM2SHORT(lastfitem,TRUE),
                (MPARAM)MPFROM2SHORT(MIA_CHECKED,FALSE));
    lastfitem = SHORT1FROMMP(mp1); /* Save menu item ID */
                                /* Convert menu ID to an attr */
    fore_clr = CHAR1FROMMP(mp1) - FORE_BLACK;
    hcell.baseattr &= 0xF0;        /* Reset foreground attribute */
    hcell.baseattr |= fore_clr;    /* Set new foreground attribute */
} /* endif */

                                /* If the second menu command is */
                                /* selected */
if (((SHORT1FROMMP(mp1)) > IID_BACKGROUND) &&
    ((SHORT1FROMMP(mp1)) < IID_EXTATTRIBUTE))
{
                                /* Get menu handle */
    wnhnd = WinWindowFromID(hwndFrame, FID_MENU);
                                /* Uncheck the previously */
                                /* selected menu item */

    WinSendMsg(wnhnd,MM_SETITEMATTR,
                (MPARAM)MPFROM2SHORT(lastbitem,TRUE),
                (MPARAM)MPFROM2SHORT(MIA_CHECKED,FALSE));
    lastbitem = SHORT1FROMMP(mp1); /* Save menu item ID */
}

```

```

/* Convert menu ID to an attr */
back_clr = (CHAR1FROMMP(mp1) - BACK_BLACK) << 4;
chcell.baseattr &= 0x0F; /* Reset background attribute */
chcell.baseattr |= back_clr; /* Set new background attribute */
} /* endif */
switch( SHORT1FROMMP( mp1 ) ) /* Extract the command value */
{
    case 301: /* Extended attr under line */
                /* Set or reset the underline */
                /* attribute */
        chcell.extattr = (chcell.extattr & 0x80) ? (chcell.extattr & 0x7F) :
            (chcell.extattr | 0x80);

        VioReadCharStr(&chcell.chr, /* Read char from cursor pos */
            &buf1,
            389,
            cursor_row,
            cursor_col,
            hvps);
        VioWrtNCell((PBYTE)&chcell, /* Rewrite char using new attrs */
            1, /* Repeat count */
            cursor_row, /* Starting row position for output */
            cursor_col, /* Starting col position for output */
            hvps ); /* Video handle */
        break;
    case 302: /* Extended attribute reverse video */
                /* Set or reset the reverse video */
                /* attribute */
        chcell.extattr = (chcell.extattr & 0x40) ? (chcell.extattr & 0xBF) :
            (chcell.extattr | 0x40);

        VioReadCharStr(&chcell.chr, /* Read character from cursor pos */
            &buf1,
            cursor_row,
            cursor_col,
            hvps);
        VioWrtNCell((PBYTE)&chcell, /* Rewrite char using new attrs */
            1, /* Repeat count */
            cursor_row, /* Starting row position for output */
            cursor_col, /* Starting col position for output */
            hvps ); /* Video handle */
        break;
}

/* Get menu handle */
wnhnd = WinWindowFromID(hwndFrame, FID_MENU);
/* Get current check state of menu item */
attrstate = (USHORT)WinSendMsg(wnhnd, MM_QUERYITEMATTR,
    (MPARAM)MPFROM2SHORT(SHORT1FROMMP(mp1), TRUE),

```

```

(MPARAM)MPFROMSHORT(MIA_CHECKED));
/* Check or uncheck menu item */
WinSendMsg(wnhnd,MM_SETITEMATTR,
(MPARAM)MPFROM2SHORT(SHORT1FROMMP(mp1),TRUE),
(MPARAM)MPFROM2SHORT(MIA_CHECKED,
attrstate ? MIA_CHECKED : MIA_CHECKED));
break;
case WM_SIZE:
/* Call avio default window proc */
/* to maintain PS and window origin*/
/* alignment */
WinDefAVioWindowProc( hWnd, msg, mp1, mp2 );
/* Get new client window size */
WinQueryWindowRect(hWnd, (PRECTL)&rect);
/* Get character size */
VioGetDeviceCellSize( &cell_height, &cell_width, hvps );
/* Determine how many columns can */
/* fit in the window */
columns = ((SHORT)rect.xRight / cell_width);
/* Determine how many rows can fit */
/* in the window */
rows = ((SHORT)rect.yTop / cell_height);
/* If window height is increased */
if (rows > old_rows)
{
vis_row -= rows - old_rows; /* Increase number of visible rows */
if (vis_row < 0) /* if at the top of the window */
{
vis_row = 0; /* Keep the top row visible */
} /* endif */
}
else
{
/* Window height is decreased so */
/* update cursor variables */
WinSendMsg(hWnd,WM_VSCROLL,
(MPARAM)FID_VERTSCROLL,
(MPARAM)MPFROM2SHORT(FALSE,SB_ENDSCROLL));
} /* endif */
if (columns > old_columns) /* If window width is increased, */
{
/* increase number of visible cols */
vis_col -= columns - old_columns;
if (vis_col < 0) /* If at the start of a line, */
{
vis_col = 0; /* keep the left column visible */
} /* endif */
}
}

```



```

else
{
    /* Window width is decreased so */
    /* update cursor variables */
    WinSendMsg(hWnd,WM_HSCROLL,
        (MPARAM)FID_HORZSCROLL,
        (MPARAM)MPFROM2SHORT(FALSE,SB_ENDSCROLL));
} /* endif */
old_rows = rows; /* Save rows variable */
old_columns = columns; /* Save columns variable */
VioSetOrg(vis_row, /* Set PS origin to row and */
    vis_col, /* column */
    hvps);
break;
case WM_PAINT:
    hps = WinBeginPaint( hWnd, (HPS)hvps, (PRECTL)&rect);
    /* Blank the area to be updated */
    WinFillRect( (HPS)hvps, (PRECTL)&rect, (ULONG)CLR_BLACK);
    /* Update the window */
    VioShowPS( rows + 1, columns + 1, (vis_row*132) + vis_col, hvps );
    WinEndPaint((HPS)hvps );
break;
case WM_CHAR:
    if (!(SHORT1FROMMP(mp1) & KC_KEYUP)) /* If key is down, */
    {
        switch (SHORT2FROMMP(mp2))
        {
            case VK_LEFT: /* left arrow key pressed */
                /* Send a line left message to */
                /* the horizontal scroll bar */
                WinSendMsg(hWnd,WM_HSCROLL,
                    (MPARAM)FID_HORZSCROLL,
                    (MPARAM)MPFROM2SHORT(FALSE,SB_LINELEFT));
                /* Send an end scroll message */
                /* to the horizontal scroll bar */
                WinSendMsg(hWnd,WM_HSCROLL,
                    (MPARAM)FID_HORZSCROLL,
                    (MPARAM)MPFROM2SHORT(FALSE,SB_ENDSCROLL));
            break;
            case VK_RIGHT: /* Right arrow key pressed */
                /* Send a line left message to */
                /* the horizontal scroll bar */
                WinSendMsg(hWnd,WM_HSCROLL,
                    (MPARAM)FID_HORZSCROLL,
                    (MPARAM)MPFROM2SHORT(FALSE,SB_LINERIGHT));
                /* Send an end scroll message */
                /* to the horizontal scroll bar */

```

```

WinSendMsg(hWnd,WM_HSCROLL,
            (MPARAM)FID_HORIZSCROLL,
            (MPARAM)MPFROM2SHORT(FALSE,SB_ENDSCROLL));

break;
case VK_UP:
            /* Up arrow key pressed */
            /* Send a line up message to */
            /* the vertical scroll bar */
            WinSendMsg(hWnd,WM_VSCROLL,
                (MPARAM)FID_VERTSCROLL,
                (MPARAM)MPFROM2SHORT(FALSE,SB_LINEUP));
            /* Send an end scroll message */
            /* to the horizontal scroll bar */
            WinSendMsg(hWnd,WM_VSCROLL,
                (MPARAM)FID_VERTSCROLL,
                (MPARAM)MPFROM2SHORT(FALSE,SB_ENDSCROLL));

break;
case VK_DOWN:
            /* Down arrow key pressed */
            /* Send a line down message to */
            /* the vertical scroll bar */
            WinSendMsg(hWnd,WM_VSCROLL,
                (MPARAM)FID_VERTSCROLL,
                (MPARAM)MPFROM2SHORT(FALSE,SB_LINEDOWN));
            /* Send an end scroll message */
            /* to the horizontal scroll bar */
            WinSendMsg(hWnd,WM_VSCROLL,
                (MPARAM)FID_VERTSCROLL,
                (MPARAM)MPFROM2SHORT(FALSE,SB_ENDSCROLL));

break;
case VK_PAGEUP:
            /* Page up key pressed */
            /* Send a page up message to */
            /* the vertical scroll bar */
            WinSendMsg(hWnd,WM_VSCROLL,
                (MPARAM)FID_VERTSCROLL,
                (MPARAM)MPFROM2SHORT(FALSE,SB_PAGEUP));
            /* Send an end scroll message */
            /* to the horizontal scroll bar */
            WinSendMsg(hWnd,WM_VSCROLL,
                (MPARAM)FID_VERTSCROLL,
                (MPARAM)MPFROM2SHORT(FALSE,SB_ENDSCROLL));

break;
case VK_PAGEDOWN:
            /* Page down key pressed */
            /* Send a page down message to */
            /* the vertical scroll bar */
            WinSendMsg(hWnd,WM_VSCROLL,
                (MPARAM)FID_VERTSCROLL,
                (MPARAM)MPFROM2SHORT(FALSE,SB_PAGEDOWN));
            /* Send an end scroll message */
            /* to the horizontal scroll bar */

```

```

WinSendMsg(hWnd,WM_VSCROLL,
            (MPARAM)FID_VERTSCROLL,
            (MPARAM)MPFROM2SHORT(FALSE,SB_ENDSCROLL));

break;
default:
break;
}
} /* endif */
break;

case WM_HSCROLL:                                /* Window scrolled horizontally */
switch (SHORT2FROMMP(mp2))
{
case SB_LINELEFT:                             /* Scroll left one character */
cursor_col--;                                /* Decrement cursor column by 1 */
if (cursor_col < 0)                          /* If cursor position is under 0 */
{
cursor_col = 0;                             /* Reset cursor col to line start */
} /* endif */
break;
case SB_PAGELEFT:                             /* Scroll left 10 characters */
cursor_col -= 10;                          /* Decrement cursor column by 10 */
if (cursor_col < 0)                          /* If cursor col is less than 0 */
{
cursor_col = 0;                             /* Reset cursor col to line start */
} /* endif */
break;
case SB_LINERIGHT:                            /* Scroll right one character */
cursor_col++;                              /* Increment cursor column by 1 */
break;
case SB_PAGERIGHT:                           /* Scroll right 10 characters */
cursor_col += 10;                          /* Increment cursor column by 10 */
break;
case SB_ENDSCROLL:                           /* Horizontal scrolling complete */
if (cursor_col > 131)                       /* If at the end of the line, */
{
cursor_col = 131;                          /* set cursor to end of the line */
vis_col = 131 - columns;                  /* Set new visible column start */
VioSetOrg(vis_row,                        /* Set PS origin to row zero, */
           vis_col,                      /* column zero */
           hvps);
}
if (cursor_col == 0)                        /* If cursor column is zero, */
{
vis_col = 0;                              /* ensure first col is visible */
VioSetOrg(vis_row,                      /* Set PS origin to row zero */
           vis_col,                    /* and current column */
           hvps);
}
}

```

```

/* Calculate new horizontal */
/* slider position */
hsldrpos = ((float)cursor_col / 131) * 100;
/* Get scroll bar handle */
wnhnd = WinWindowFromID(hwndFrame, SHORT1FROMMP(mp1));
/* Set new slider position */
WinPostMsg(wnhnd, SBM_SETPOS, (MPARAM)hsldrpos, 0L);
break;
case SB_SLIDERPOSITION: /* Horizontal slider moved directly */
/* Get scroll bar handle */
wnhnd = WinWindowFromID(hwndFrame, SHORT1FROMMP(mp1));
/* Set new slider position */
WinPostMsg(wnhnd, SBM_SETPOS, (MPARAM)SHORT1FROMMP(mp2), 0L);
hsldrpos = SHORT1FROMMP(mp2); /* Save slider position */
/* Calculate new column number */
cursor_col = 131 * ((float)hsldrpos / 100);
break;
default:
break;
} /* endswitch */

/* If cursor is right of window, */
if (cursor_col > (columns + vis_col) - 1)
{
/* set new visible column */
vis_col = cursor_col - (columns - 1);
VioSetOrg(vis_row, /* Set PS origin to visible */
vis_col, /* row and column number */
hvps);
} /* endif */

/* Cursor is left of the window */
if (cursor_col < vis_col)
{
vis_col = cursor_col; /* Visible column is cursor col */
VioSetOrg(vis_row, /* Set PS origin to visible */
vis_col, /* row and column number */
hvps);
} /* endif */
VioSetCurPos(cursor_row, /* Set new cursor row */
cursor_col,
hvps);
VioReadCharStr(&chcell.chr, /* Read character from cursor pos */
&buf1,
cursor_row,
cursor_col,
hvps);
VioWrtnCell((PBYTE)&chcell, /* Cell to be written */
1, /* Repeat count */
cursor_row, /* Starting row pos for output */

```

```

        cursor_col,      /* Starting column pos for output */
        hvps );         /* Video handle */
break;
case WM_VSCROLL:        /* Window scrolled vertically */
    switch (SHORT2FROMMP(mp2))
    {
        case SB_LINEUP:    /* Window scrolled up one line */
            cursor_row--;   /* Decrement cursor row */
            if (cursor_row < 0) /* If cursor row is less than 0, */
            {
                cursor_row = 0; /* set cursor to top of window */
            } /* endif */
            break;
        case SB_PAGEUP:    /* Window scrolled up one page */
            cursor_row -= 10; /* Decrement cursor pos by 10 */
            if (cursor_row < 0) /* Cursor is less than zero */
            {
                cursor_row = 0; /* Set cursor to top of window */
            } /* endif */
            break;
        case SB_LINEDOWN:  /* Window scrolled down one line */
            cursor_row++;    /* Increment cursor row by one */
            break;
        case SB_PAGEDOWN:  /* Window scrolled down one page */
            cursor_row += 10; /* Set cursor down ten lines */
            break;
        case SB_ENDSCROLL: /* Vertical scrolling complete */
            if (cursor_row > 35) /* If cursor is beyond last line, */
            {
                cursor_row = 35; /* set cursor row to last line */
                vis_row = (35 - (rows - 1)); /* Set new visible row */
                VioSetOrg(vis_row, /* Set PS origin to visible */
                           vis_col, /* row and column number */
                           hvps);
            }
            if (cursor_row == 0) /* If cursor is at top of window, */
            {
                vis_row = 0; /* visible row is at top of window */
                VioSetOrg(vis_row, /* Set PS origin to visible */
                           vis_col, /* row and column number */
                           hvps);
            }
            /* Establish new slider position */
            vsldrpos = ((float)cursor_row / 35) * 100;
            /* Get scroll bar window handle */
            wnhnd = WinWindowFromID(hwndFrame, SHORT1FROMMP(mp1));
            /* Set new slider position */
            WinPostMsg(wnhnd, SBM_SETPOS, (MPARAM)vsldrpos, 0L);

```

```

break;
case SB_SLIDERPOSITION:      /* Vertical slider moved directly */
                             /* Get scroll bar window handle */
    wnhnd = WinWindowFromID(hwndFrame, SHORT1FROMMP(mp1));
                             /* Update slider position */
    WinPostMsg(wnhnd, SBM_SETPOS, (MPARAM)SHORT1FROMMP(mp2), 0L);
    vsldrpos = SHORT1FROMMP(mp2); /* Save slider position */
                             /* Calculate new cursor row */
    cursor_row = 35 * ((float)vsldrpos / 100);
break;
default:
break;
} /* endswitch */

                             /* If cursor is below window bottom */
if (cursor_row > (vis_row + rows) )
{
    vis_row = cursor_row - (rows - 1); /* set new cursor row */
    VioSetOrg(vis_row,                /* Set PS origin to new row */
              vis_col,                /* and column */
              hvps);
} /* endif */
if (cursor_row < vis_row)          /* If cursor is above window top */
{
    vis_row = cursor_row;
    VioSetOrg(vis_row,                /* set PS origin to new row */
              vis_col,                /* and column */
              hvps);
} /* endif */
VioSetCurPos(cursor_row,           /* Set new cursor row */
              cursor_col,
              hvps);
VioReadCharStr(&chcell.chr,        /* Read character from cursor pos */
              &bufl,
              cursor_row,
              cursor_col,
              hvps);
VioWrtnCell((PBYTE)&chcell,        /* Rewrite char to the window */
            1,                      /* Repeat count */
            cursor_row,             /* Starting row pos for output */
            cursor_col,             /* Starting col pos for output */
            hvps );                /* Video handle */

break;
default:

/*****
/* Everything else comes here. This call MUST exist in your
/* window procedure
*****/

```

```

        return( (MRESULT)WinDefWindowProc( hWnd, msg, mp1, mp2 ));
        break;
    }
    return( 0L );
}

/***** End of Window procedure *****/
/***** End of source file *****/

```

This program starts by initializing with the Presentation Manager system. The program then registers and creates a standard window. When the window is registered, the CS_SIZEREDRAW style is used so that the window procedure has a chance to repaint the window whenever the size of the window is changed.

The icon and accelerator controls are removed from the standard control because they are not needed for this example. The vertical and horizontal scroll bars are then added to the frame controls. A standard window is then created. The rest of the main loop remains similar to other Presentation Manager programs.

The messages handled in the AVIOPGM.C program include the create (WM_CREATE), command (WM_COMMAND), size (WM_SIZE), paint (WM_PAINT), keystroke (WM_CHAR), horizontal scroll (WM_HSCROLL), and vertical scroll (WM_VSCROLL) messages.

When the window is created, the window procedure will receive a WM_CREATE message before the window is shown. At this time, a window device context is created. Then an AVIO presentation space to be associated with the window device context is created.

The presentation space created is 132 columns $\times$ 36 lines and uses the extended attributes. WinSetWindowPos is then used to establish the initial size and position of the window. The presentation space associated with the device context uses VioAssociate. Once the AVIO presentation space is associated with the window device context, the window is referenced as an AVIO window.

The presentation space is initialized with some fixed data to demonstrate the use of the extended attribute presentation space. The cursor is set in the upper left corner of the window as the final initialization for the window.

The command message processes user selections from the menu. You can change the foreground color attribute, background color attribute, and the extended attribute. If the foreground or background color is changed, the program unchecks the previously checked menu item and saves the current selection. The menu item ID is then converted to an attribute and stored in the base attribute.

If either the underline or the reverse video attribute in the extended attribute is changed, it is stored in the extended attribute. The character at the current cursor location is then read and written back to the screen to reflect the change. The last selected menu item is then checked so that the next time you select the menu you will see what selection you made before.

When the window size and position is established during creation, a `WM_SIZE` message is sent to the window procedure. This message is also received when the user changes the dimensions of the window with either the keyboard or mouse.

First, the default AVIO window procedure is called. This is done so that the window is aligned with the presentation space. If the default AVIO window procedure was not called, setting the presentation space origin functions erratically. The dimensions of the client area are then obtained by calling `WinQueryWindowRect`.

The character size is then obtained using the `VioGetDeviceCellSize` function. The number of characters on a line is determined by dividing the window width by the character width. The number of lines in the window is determined by dividing the height of the window by the character height. If the window size is increased, more lines can be displayed in the window. If the window size is decreased, the window is updated and the cursor resumes the same position it had before. The last step is resetting the origin of the presentation space position in the upper left corner of the window.

After the window is resized, a `WM_PAINT` message is received, because the window was created with the `CS_SIZEREDRAW` style. The window is cleared by calling `WinFillRect`, and the data is then displayed by calling `VioShowPS`.

The window can be scrolled using the mouse or the keyboard. When one of the cursor movement keys, `PgUp`, or `PgDn` is pressed, a `WM_CHAR` message is received by the window procedure. A message is received when the key is pressed, and a second message is received when the key is released. To avoid acting on both, only those messages as a result of key down transitions are reacted to. This is indicated when a `WM_CHAR` message is received with the `KC_KEYUP` bit turned off.

The cursor left (left arrow) key or the cursor right (right arrow) key is sent to the horizontal scroll bar. The cursor up (up arrow), the cursor down (down arrow), the `PgUp`, or the `PgDn` key is sent to the vertical scroll bar. The `WinSendMsg` function is used to send the appropriate message to the scroll bar selected. Once the message is sent to the scroll bar, the end scroll message is next sent to the scroll bar to update the slider position.

When the mouse is used to scroll the window, the scrolling is indicated by placing the pointer on a part of the horizontal/vertical scroll bar and pressing mouse button one. More direct scrolling can be achieved by placing the pointer on the scroll bar slider, pressing and holding button one, then moving the slider to a new position and releasing mouse button one. When the horizontal scroll bar is manipulated, a `WM_HSCROLL` message is received. When the vertical scroll bar is manipulated, a `WM_VSCROLL` message is received.

When scrolling is performed, the variables holding the scrolling position are updated. Based on the cursor position, the presentation space coordinates that hold the position visible in the upper left corner of the window are updated. If the cursor is moved, the slider reflects the relative position of the cursor in the presentation space. If the slider is moved directly, the cursor is moved to the relative position in the presentation space.

Scrolling completion is indicated when the scroll message is SB_ENDSCROLL. When the slider bar is directly manipulated, the scroll message is SB_SLIDERPOSITION.

You can now see why an AVIO program is a Presentation Manager program. The basic program is similar to that of a full-function Presentation Manager program. The use of the base video APIs to display data, however, restricts the program to using only text. The intent of this program is to show how the different attributes would be used and demonstrate the support of scroll bars, using the mouse and keyboard.

To complete the program, the DEF, H, and RC files are given here, along with the CMD file used to build the program. Using these tools, you can rebuild the entire application.

```
/****** This is the file AVIOPGM.H *****/
#define IID_FOREGROUND 100
#define FORE_BLACK 101
#define FORE_BLUE 110
#define FORE_RED 113
#define FORE_PINK 114
#define FORE_GREEN 111
#define FORE_CYAN 112
#define FORE_YELLOW 115
#define FORE_WHITE 116
#define FORE_DKGRAY 109
#define FORE_DKBLUE 102
#define FORE_DKRED 105
#define FORE_DKPINK 106
#define FORE_DKGREEN 103
#define FORE_DKCYAN 104
#define FORE_BROWN 107
#define FORE_PLGRAY 108
#define IID_BACKGROUND 200
#define BACK_BLACK 201
#define BACK_BLUE 210
#define BACK_RED 213
#define BACK_PINK 214
#define BACK_GREEN 211
#define BACK_CYAN 212
#define BACK_YELLOW 215
#define BACK_WHITE 216
#define BACK_DKGRAY 209
#define BACK_DKBLUE 202
#define BACK_DKRED 205
#define BACK_DKPINK 206
#define BACK_DKGREEN 203
#define BACK_DKCYAN 204
#define BACK_BROWN 207
#define BACK_PLGRAY 208
#define IID_EXTATTRIBUTE 300
```

```

/***** This is the file AVIOPGM.RC *****/
#include "aviopgm.h"
MENU 1 MOVEABLE DISCARDABLE
BEGIN
    SUBMENU "Foreground", IID_FOREGROUND
    BEGIN
        MENUITEM "Black",      FORE_BLACK
        MENUITEM "Blue",       FORE_BLUE
        MENUITEM "Red",        FORE_RED
        MENUITEM "Pink",       FORE_PINK
        MENUITEM "Green",      FORE_GREEN
        MENUITEM "Cyan",       FORE_CYAN
        MENUITEM "Yellow",     FORE_YELLOW
        MENUITEM "White",      FORE_WHITE
        MENUITEM "Dark Gray",  FORE_DKGRAY
        MENUITEM "Dark Blue",  FORE_DKBLUE
        MENUITEM "Dark Red",   FORE_DKRED
        MENUITEM "Dark Pink",  FORE_DKPINK
        MENUITEM "Dark Green", FORE_DKGREEN
        MENUITEM "Dark Cyan",  FORE_DKCYAN
        MENUITEM "Dark Brown", FORE_BROWN
        MENUITEM "Pale Gray",  FORE_PLGRAY
    END
    SUBMENU "Background", IID_BACKGROUND
    BEGIN
        MENUITEM "Black",      BACK_BLACK
        MENUITEM "Blue",       BACK_BLUE
        MENUITEM "Red",        BACK_RED
        MENUITEM "Pink",       BACK_PINK
        MENUITEM "Green",      BACK_GREEN
        MENUITEM "Cyan",       BACK_CYAN
        MENUITEM "Yellow",     BACK_YELLOW
        MENUITEM "White",      BACK_WHITE
        MENUITEM "Dark Gray",  BACK_DKGRAY
        MENUITEM "Dark Blue",  BACK_DKBLUE
        MENUITEM "Dark Red",   BACK_DKRED
        MENUITEM "Dark Pink",  BACK_DKPINK
        MENUITEM "Dark Green", BACK_DKGREEN
        MENUITEM "Dark Cyan",  BACK_DKCYAN
        MENUITEM "Dark Brown", BACK_BROWN
        MENUITEM "Pale Gray",  BACK_PLGRAY
    END
    SUBMENU "Extended attribute", 300
    BEGIN
        MENUITEM "Underline", 301
        MENUITEM "Reverse video", 302
    END
END

```

```

**/
/***** This is the file AVIOPGM.DEF *****/
NAME      aviopgm
DESCRIPTION 'OS/2 Presentation Manager Sample'

CODE      MOVEABLE
DATA      MOVEABLE MULTIPLE

HEAPSIZE   1024          ; Must be nonzero to use Local memory manager
STACKSIZE  4096          ; Must be nonzero for SS == DS
                                   ; Suggest 4k as minimum stacksize

EXPORTS
    AvioWndProc @1      ; Must export all procedures called by Pres. Mgr.
                                   ; (ordinal numbers use less resident memory)

/***** This is the CMD file to build the AVIO program *****/
cl -Zp -c -AlfuL -Gsc aviopgm.c
link /CO aviopgm,,/NOD os2 llibce,aviopgm;
rc aviopgm
```

Window Templates

You have seen how simple windows can be created using `WinCreateWindow`. You have also seen how to create standard windows using the `WinCreateStdWindow` function. An alternative method of creating windows for your application is with window templates. Window templates enable you to design the appearance of your windows in parallel with your application development and keep these designs separate from the source code. The use of window templates also enables you to treat windows like panels. The same function can be used to load different windows, just like you do for dialogs.

Creating and Using Window Templates

Window templates are created with the resource definition file, much the same way you create menus. The changes required so your program can use templates are discussed in this chapter. You will then see the additions that are needed for the resource file to enable the use of these window templates.

When you use templates, the window class must be registered the same way as if you used the `WinCreateWindow` or `WinCreateStdWindow` APIs to create the window. The window is created from the template with the `WinLoadDlg` API instead of the `WinCreateWindow` or the `WinCreateStdWindow` API. The code to perform this function is

```

/* Load window template */
hwndClient = WinLoadDlg(HWND_DESKTOP, /* Window parent      */
                        HWND_DESKTOP, /* Window owner       */
                        NULL,         /* Not a dialog        */
                        NULL,         /* Not loaded from a DLL */
                        1,            /* Window ID of one    */
                        (PVOID)0);    /* No creation parameters */

```

Once this function is executed, the window template is loaded, and a WM_CREATE is sent to the window procedure called to initialize the window. If the window is going to have a menu, it may be specified right inside the window template. It could also be loaded explicitly using WinLoadMenu.

If the latter method is used, the menu is loaded using the WinQueryWindow function to get the frame window handle and then using the handle to load the menu by calling WinLoadMenu. This is accomplished with code like the following:

```

/* Get the frame window handle */
hwndFrame = WinQueryWindow(hWnd, /* Client window handle */
                           QW_PARENT, /* Get the frame window handle */
                           FALSE); /* Do not lock the window */

wnhnd = WinLoadMenu(hwndFrame, /* Frame window handle */
                    NULL,      /* Menus are part of the .EXE */
                    1);        /* Menu ID */

```

Now that the program is modified to use window templates, take a look at what is involved to define the template.

First, the method that explicitly loads the menu is described. The resource definition file for this method is

```

WINDOWTEMPLATE 1
BEGIN
    FRAME "Sample AVIO Window", 1, 0, 50, 500, 300
    CTLDATA 0x00000CFF
    BEGIN
        WINDOW "", FID_CLIENT, 0, 0, 0, 0, "AVIOWIN"
    END
END

```

The window is given an ID of 1. The frame window is specified with the FRAME statement. The frame control flags, FCF, are specified as 0x00000CFF in the CTLDATA statement. Using this method, if you specify the flags using the OR operator to set the bits you will get an error from the resource compiler. Take a look at the flags you wish to use and then set the frame control flag value of the template to be the logical ORing of those values.

WINDOW TEMPLATES

For example, 0x0000CFF is the result of ORing the following flags: FCF_TITLEBAR, FCF_SYSMENU, FCF_MENU, FCF_SIZEBORDER, FCF_MINBUTTON, FCF_MAXBUTTON, FCF_VERTSCROLL, FCF_HORZSCROLL, FCF_SHELLPOSITION, and FCF_TASKLIST.

The client window is created by specifying the window statement within the template. It is given a window ID of FID_CLIENT and the class AVIOWIN is used as it was registered in the program. The size and position are specified as zero. Following is a sample of the window template used.

The resource file used to create the menu without having to go through it with WinLoadMenu is

WINDOWTEMPLATE 1

BEGIN

```
FRAME "Sample AVIO Window", 1, 0, 50, 300, 200, CS_SIZEDRAW,
    FCF_SYSMENU ! FCF_TITLEBAR ! FCF_MINMAX !
    FCF_MENU ! FCF_SIZEBORDER !
    FCF_VERTSCROLL ! FCF_HORZSCROLL
```

BEGIN

```
CONTROL "", FID_MENU, 0, 191, 300, 10, WC_MENU, MS_ACTIONBAR ! WS_VISIBLE
CTLDATA MENU
```

BEGIN

SUBMENU "Foreground", IID_FOREGROUND

BEGIN

```
MENUITEM "Black", FORE_BLACK
MENUITEM "Blue", FORE_BLUE
MENUITEM "Red", FORE_RED
MENUITEM "Pink", FORE_PINK
MENUITEM "Green", FORE_GREEN
MENUITEM "Cyan", FORE_CYAN
MENUITEM "Yellow", FORE_YELLOW
MENUITEM "White", FORE_WHITE
MENUITEM "Dark Gray", FORE_DKGRAY
MENUITEM "Dark Blue", FORE_DKBLUE
MENUITEM "Dark Red", FORE_DKRED
MENUITEM "Dark Pink", FORE_DKPINK
MENUITEM "Dark Green", FORE_DKGREEN
MENUITEM "Dark Cyan", FORE_DKCYAN
MENUITEM "Dark Brown", FORE_BROWN
MENUITEM "Pale Gray", FORE_PLGRAY
```

END

SUBMENU "Background", IID_BACKGROUND

BEGIN

```
MENUITEM "Black", BACK_BLACK
MENUITEM "Blue", BACK_BLUE
MENUITEM "Red", BACK_RED
MENUITEM "Pink", BACK_PINK
MENUITEM "Green", BACK_GREEN
```

```

MENUITEM "Cyan",      BACK_CYAN
MENUITEM "Yellow",    BACK_YELLOW
MENUITEM "White",     BACK_WHITE
MENUITEM "Dark Gray", BACK_DKGRAY
MENUITEM "Dark Blue", BACK_DKBLUE
MENUITEM "Dark Red",  BACK_DKRED
MENUITEM "Dark Pink", BACK_DKPINK
MENUITEM "Dark Green", BACK_DKGREEN
MENUITEM "Dark Cyan", BACK_DKCYAN
MENUITEM "Dark Brown", BACK_BROWN
MENUITEM "Pale Gray", BACK_PLGRAY
END
SUBMENU "Extended attribute", 300
BEGIN
    MENUITEM "Underline",      301
    MENUITEM "Reverse video",  302
END
END
WINDOW "", FID_CLIENT, 0, 0, 300, 190, "AVIOWIN"
END
MENU 1 MOVEABLE DISCARDABLE
BEGIN
    SUBMENU "Foreground", IID_FOREGROUND
    BEGIN
        MENUITEM "Black",      FORE_BLACK
        MENUITEM "Blue",       FORE_BLUE
        MENUITEM "Red",        FORE_RED
        MENUITEM "Pink",       FORE_PINK
        MENUITEM "Green",      FORE_GREEN
        MENUITEM "Cyan",       FORE_CYAN
        MENUITEM "Yellow",     FORE_YELLOW
        MENUITEM "White",      FORE_WHITE
        MENUITEM "Dark Gray",  FORE_DKGRAY
        MENUITEM "Dark Blue",  FORE_DKBLUE
        MENUITEM "Dark Red",   FORE_DKRED
        MENUITEM "Dark Pink",  FORE_DKPINK
        MENUITEM "Dark Green", FORE_DKGREEN
        MENUITEM "Dark Cyan",  FORE_DKCYAN
        MENUITEM "Dark Brown", FORE_BROWN
        MENUITEM "Pale Gray",  FORE_PLGRAY
    END
    SUBMENU "Background", IID_BACKGROUND
    BEGIN
        MENUITEM "Black",      BACK_BLACK
        MENUITEM "Blue",       BACK_BLUE
        MENUITEM "Red",        BACK_RED
        MENUITEM "Pink",       BACK_PINK

```

```

MENUITEM "Green",      BACK_GREEN
MENUITEM "Cyan",       BACK_CYAN
MENUITEM "Yellow",     BACK_YELLOW
MENUITEM "White",      BACK_WHITE
MENUITEM "Dark Gray",  BACK_DKGRAY
MENUITEM "Dark Blue",  BACK_DKBLUE
MENUITEM "Dark Red",   BACK_DKRED
MENUITEM "Dark Pink",  BACK_DKPINK
MENUITEM "Dark Green", BACK_DKGREEN
MENUITEM "Dark Cyan",  BACK_DKCYAN
MENUITEM "Dark Brown", BACK_BROWN
MENUITEM "Pale Gray",  BACK_PLGRAY
END
SUBMENU "Extended attribute", 300
BEGIN
    MENUITEM "Underline",      301
    MENUITEM "Reverse video",  302
END
END

```

As you can see, the entire menu may be defined directly in the window template definition. Using the first method shown, you need to explicitly create the menu to be used in the window. In the second instance, the menu is created when the window is. The method you use depends on the functions you wish to perform with the menu.

The remainder of the program and the resource files remain unchanged. Although the sample shown had the template and resources as a part of the program, they could have been just as easily loaded from a dynamic link resource library.

All these items may be placed into the AVIO sample program given in the previous chapter. When this is done, you (as a user) would not know which method was used to create the window. This feature is provided to enable you to structure code in many different ways depending on the needs of the application.

Summary

Window templates are provided as an aid for designing and structuring your application. They enable you to separate the user interface from the application's function. Window templates make it easier to fine-tune the user interface. Small changes such as changing the appearance of menu items can be made without rebuilding the entire program. Rebuilding the program is necessary only when the function changes.

A

Application Programming Interfaces

The following is a list of all the APIs available to OS/2 Version 1.2 programs. The list is in alphabetical order.

This list is not meant to replace the technical reference manuals but to give a concise description of each function call and the list of parameters expected by each call. For more detailed information, consult the technical reference manuals.

DevCloseDC Returns an error status or a handle to a metafile (HMF) when you are working with a metafile.

This function closes a previously opened device context.

Parameter	Description
HDC	Handle of the device context to be closed

DevEscape Returns the status (LONG).

This function sends an escape code to a device driver. Escape codes must be understood by the device driver.

Parameters	Description
HDC	Handle to a previously opened device context
LONG	The escape code to be sent
LONG	The number of bytes to be sent
PBYTE	The actual data to be sent
PLONG	Size of the output buffer
PBYTE	Buffer to receive any returned data

DevOpenDC Returns an error status or the handle to a device context (HDC).

This function will open (create) a device context for use by a process.

Parameters	Description
HAB	Handle to an anchor block returned by WinInitialize
LONG	The type of device context to be opened
PSZ	The device information token for the new DC
LONG	The number of elements in the DEVOPENDATA structure
PDEVOPENDATA	Pointer to the DEVOPENDATA structure
HDC	Handle of a compatible DC (if desired)

DevPostDeviceModes Returns status or data length (LONG).

This function is used to have the user directly specify control information. When this call is issued the device driver will display a dialog box to obtain the information from the user. This control information is later used as the driver data for the DevOpenDC call.

Parameters	Description
HAB	Handle to an anchor block returned by WinInitialize
PDRVDATA	Address of information used for DevOpenDC driver data
PSZ	Address of the device driver name
PSZ	Address of the device type name
PSZ	Address of the device name
ULONG	Options used to display the dialog box

DevQueryCaps Returns pass or fail status (BOOL).

This function is used to determine the device features. You must indicate how much information you want and where the information is to be returned.

Parameters	Description
HDC	Handle of the device context to be queried
LONG	First feature for which information is to be returned
LONG	Number of features to be returned
PLONG	Address of array used to hold the features returned

DevQueryDeviceNames Returns pass or fail status (BOOL).

This function enables you to query a device driver and determine the number of devices the driver supports. You can also determine the description of each device and the data type supported by each device.

Parameters	Description
HAB	Handle to an anchor block returned by WinInitialize
PSZ	The name of the device driver to be queried
PLONG	Count of device names and their descriptions returned
PSTR32	Address of array holding the device names
PSTR64	Address of array holding the device descriptions
PLONG	Count of the supported data types
PSTR16	Address of the array holding the data types

DevQueryHardcopyCaps Returns an error status or the number of forms returned (LONG).

This function returns the number of hardcopy limits referenced by the device context handle. The information will be returned in the array of structures starting with the one requested first.

Parameters	Description
HDC	Handle of the device context to be queried
LONG	The forms code to start with
LONG	Count of forms to be queried
PHCINFO	Address of array of structures in which the results will be placed

DosAllocHuge Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to allocate memory greater than 64KB. You will specify the size range and the controlling flags. A selector to the allocated memory will be returned.

Parameters	Description
USHORT	Number of 64KB segments
USHORT	Number of bytes in last segment, if less than 64KB
PSEL	The address of the selector for first segment
USHORT	Upper limit of segment count
USHORT	Memory allocation control (share/discard/reallocate) flags

DosAllocSeg Failure returns an error code; otherwise, zero is returned (USHORT).

This function allocates a memory segment of up to 64KB. The requested size is passed as the first parameter, and the control information is passed as the third parameter. If the call is successful, a selector for the segment is returned in the second parameter.

Parameters	Description
USHORT	The number of bytes requested
PSEL	The address of the selector for the allocated memory
USHORT	Memory allocation control (share/discard/reallocate) flags

DosAllocShrSeg Failure returns an error code; otherwise, zero is returned (USHORT).

The call allocates a shared memory segment that is referenced by name. A pointer to the name for the shared segment is passed as the first parameter. If the call is successful, the selector for the shared memory is returned.

Parameters	Description
USHORT	The number of bytes requested
PSZ	The address of the name string
PSEL	The address of the selector for the allocated memory

DosBeep Failure returns an error code; otherwise, zero is returned (USHORT).

This function causes the generation of a sound from the speaker that is controlled by the input parameters.

Parameters	Description
USHORT	The frequency of the sound to be generated
USHORT	The number of milliseconds the sound will last

DosBufReset Failure returns an error code; otherwise, zero is returned (USHORT).

The cache buffers for a specific file are written to the disk.

Parameter	Description
HFILE	The file handle for the file whose buffers are to be written to disk

DosCallback Does not return a value (VOID).

This function allows an IOPL (ring 2) segment to call a regular (ring 3) application segment.

Parameter	Description
PFN	The address of the application segment that is being called

DosCallNmPipe Failure returns an error code; otherwise, zero is returned (USHORT).

This function allows a process to make a procedure call over a named pipe to another process.

Parameters	Description
PSZ	The address of pipe name string
PBYTE	The address of request buffer
USHORT	The request length
PBYTE	The address of response buffer
USHORT	The size of data expected
PUSHORT	The number of bytes returned
ULONG	Timeout value

DosCaseMap Failure returns an error code; otherwise, zero is returned (USHORT).

This function performs case mapping on the string referenced by the third parameter.

Parameters	Description
USHORT	The length of string to be mapped
PCOUNTRYCODE	The address of country information structure that will determine the code page used to do the mapping
PCHAR	The address of what will be mapped in place

DosChDir Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to change the current directory for the issuing process.

Parameters	Description
PSZ	The address of the directory path name
ULONG	This is reserved and must be zero

DosChgFilePtr Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used when you perform random I/O to a file. It is used to set the location (file pointer) from which I/O will be performed.

Parameters	Description
HFILE	The handle of the file being manipulated
LONG	The offset to be applied to the current pointer
USHORT	Way that the offset is to be applied from the start of the file, current location, or the end of the file
PULONG	The address of the updated file pointer

DosCLIAccess Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used by an IOPL routine to request the ability to enable and disable interrupts.

Parameter	Description
VOID	No parameters required

DosClose Failure returns an error code; otherwise, zero is returned (USHORT).
This function closes a file and invalidates the handle.

Parameter	Description
HFILE	The file handle that is to be closed

DosCloseQueue Failure returns an error code; otherwise, zero is returned (USHORT).
This function closes a queue and invalidates the handle.

Parameter	Description
HQUEUE	The queue handle that is to be closed

DosCloseSem Failure returns an error code; otherwise, zero is returned (USHORT).
This function closes a system semaphore.

Parameter	Description
HSEM	The system semaphore handle that is to be closed

DosConnectNmPipe Failure returns an error code; otherwise, zero is returned (USHORT).
This function allows a process to gain access to a named pipe.

Parameter	Description
HPIPE	The named pipe handle used to access the pipe

DosCopy Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to copy the contents of a file to a new file.

Parameters	Description
PSZ	Pointer to the source path and name string
PSZ	Pointer to the target path and name string
USHORT	The flags used to determine whether the source should be appended to or replace the target file, and to determine what to do if the file already exists
ULONG	This is a reserved parameter and must be zero

DosCreateCSAlias Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used by a process to implement read/write code segments. It passes a selector to a data segment and is returned a code segment selector that it can use to execute code in the segment.

Parameters	Description
SEL	The data segment selector containing the code to be executed
PSEL	The address of the code segment selector

DosCreateQueue Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to create a queue whose name is passed in the third parameter. The elements will be ordered as indicated by the second parameter. If the queue is created, the handle is returned in the first parameter.

Parameters	Description
PHQUEUE	The address where the queue handle is returned
USHORT	The elements will be determined as requested, first in first out, last in first out, or by priority of the writing process
PSZ	The address of the queue name string

DosCreateSem Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to create a system semaphore whose name is passed in the third parameter.

Parameters	Description
USHORT	This parameter is used to indicate whether or not exclusive ownership is required
PHSYSSEM	The address where the semaphore handle is placed
PSZ	The address of the semaphore name string

DosCreateThread Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to initiate another instance of execution (a thread) within an application. It passes the address of the routine where execution is to start and the address of the memory segment that will be used as the new thread's stack. If the thread is created the ID is returned in the second parameter.

Parameters	Description
PFNTHREAD	The starting address of the new thread
PTID	The address where the new thread ID is returned
PBYTE	Address of the thread's stack

DosCwait Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to wait for the results from an asynchronously started child process.

Parameters	Description
USHORT	This parameter is used to indicate whether the thread should wait for just the child process or the child process and all its descendants
USHORT	This parameter is used to indicate whether the thread should wait if the child process is still running
PRESULTCODES	The address where the result codes will be stored
PPID	The address where the process ID of the terminating process will be placed
PID	The process ID of the process to wait for

DosDelete Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to delete a file from a disk.

Parameters	Description
PSZ	The address of the string containing the file path and name
ULONG	This is a reserved parameter and must be zero

DosDevConfig Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to determine the system model, system type, or devices configured. The device of interest is indicated by parameter two and the information is returned in the buffer whose address is passed in parameter one.

Parameters	Description
PVOID	The address of the buffer where the information will be returned
USHORT	The indicator of the device of interest
USHORT	This is a reserved parameter and must be zero

DosDevIOCtl Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to perform device control functions from an application.

Parameters	Description
PVOID	The address of the data buffer
PVOID	The address of the command arguments
USHORT	The device control function to be performed
USHORT	The device category (class)
HFILE	The device handle returned from a previous DosOpen

DosDevIOCtl2 Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to perform device control functions from an application. It supports the specification of data length and argument list length for Version 1.2 device drivers.

Parameters	Description
PVOID	Pointer to the data buffer
USHORT	The data buffer length
PVOID	Pointer to command-specific argument list
USHORT	The length of argument list
USHORT	The device control function to be performed
USHORT	The device category (class)
HFILE	The device handle returned from a previous DosOpen

DosDisconnectNmPipe Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to close a named pipe.

Parameter	Description
HPIPE	The named pipe handle

DosDupHandle Failure returns an error code; otherwise, zero is returned (USHORT).
This function will return a new file handle in parameter two for the file whose handle is passed in parameter one.

Parameters	Description
HFILE	The current file handle
PHFILE	The address of the new file handle

DosEditName Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to transform a source string into a destination string using an editing string and OS/2 metaediting semantics.

Parameters	Description
USHORT	A value of 0x0001 specifies that OS/2 1.2 editing semantics are to be used

APPENDIX A

PSZ	Pointer to the source string
PSZ	Pointer to the editing string
PBYTE	Pointer to the destination string buffer
USHORT	The destination string buffer length

DosEnterCritSec Does not return a value (VOID).

This function is used to initiate execution serialization within a common code routine.

Parameter	Description
VOID	No parameters required

DosEnumAttribute Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to enumerate information for a specific file or subdirectory.

Parameters	Description
USHORT	The reference type—a handle or a name buffer
PVOID	Pointer to the handle or the name buffer
ULONG	The extended attribute ordinal number
PVOID	Pointer to enumeration buffer
ULONG	The enumeration buffer size
PULONG	Pointer to count of extended attributes whose information is requested on input and the system returns the count of attributes returned
ULONG	The level of information required
ULONG	This is a reserved parameter and must be zero

DosErrClass Failure returns an error code; otherwise, zero is returned (USHORT).

This function will analyze an error code returned from a previous kernel call passed in parameter one and return the analysis information in parameters two, three, and four.

Parameters	Description
USHORT	The error code to be analyzed
PUSHORT	The error code classification
PUSHORT	The recommended recovery action
PUSHORT	The part of the system that had the error

DosError Failure returns an error code; otherwise, zero is returned (USHORT).

This function allows the application to enable or disable OS/2 hard error popup or exception notification.

Parameter	Description
USHORT	Based on the value, the hard error popup or exception notification will be enabled or disabled

DosExecPgm Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to start a program as an asynchronous or synchronous child process.

Parameters	Description
PCHAR	The address of a buffer used to return additional information that would indicate why the call failed
SHORT	The length of parameter one buffer

USHORT	The execution flag, how the program should be executed
PSZ	The address of the argument string
PSZ	The address of the environment string
PRESULTCODES	The address of the structure that will get the termination codes
PSZ	The address of the program path and name string

DosExit Does not return a value (VOID).

This function is used to terminate the current thread or the process.

Parameters	Description
USHORT	The termination indicator
USHORT	The result code to save for DosCwait

DosExitCritSec Does not return a value (VOID).

This function is used to end execution serialization within a common code routine.

Parameter	Description
VOID	No parameters required

DosExitList Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to add or remove one or more routines to or from a list. This list of routines will be called when the process for which the list is established terminates.

Parameters	Description
USHORT	The request code
PFNEXITLIST	Pointer to the routine

DosFileIO Failure returns an error code; otherwise, zero is returned (USHORT).

This function enables you to combine multiple file locks/unlocks, seeks, reads, and writes into a single request.

Parameters	Description
HFILE	The file handle
PBYTE	Pointer to the command list buffer
USHORT	The length of the command list
PLONG	Pointer to where the system will return the offset of the command list that was in error

DosFileLocks Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to lock or unlock a number of bytes in a previously opened file.

Parameters	Description
HFILE	The file handle of file to lock or unlock
PLONG	The address of starting lock/unlock file offset
PLONG	The address of number of bytes to lock/unlock

DosFindClose Failure returns an error code; otherwise, zero is returned (USHORT).

This function closes a directory handle used in a find first/find next search operation.

Parameter	Description
HDIR	The directory search handle

DosFindFirst Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to find the first match in a directory search operation.

Parameters	Description
PSZ	Pointer to the file path name string
PHDIR	Pointer to the directory search handle
USHORT	The file attributes used in the search
PFILEFINDBUF	Pointer to the search buffer
USHORT	Search buffer length
PUSHORT	Pointer to search count
ULONG	This parameter is reserved and must be zero

DosFindFirst2 Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to find the first match in a directory search operation providing additional capabilities for extended attributes.

Parameters	Description
PSZ	Pointer to the file path and name string
PHDIR	Pointer to the directory search handle
USHORT	The search attribute
PVOID	The address of the result buffer
USHORT	The result buffer length
PUSHORT	Pointer to the number of entries to find on input and the address of the number of entries placed in the result buffer
USHORT	The level of information requested
ULONG	This is a reserved parameter and must be zero

DosFindNext Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to find the next matching entries in a directory search operation.

Parameters	Description
HDIR	The directory search handle
PFILEFINDBUF	Pointer to the search buffer
USHORT	Search buffer length
PUSHORT	Pointer to search count

DosFindNotifyClose Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to close a find-notify handle.

Parameter	Description
HDIR	The handle of the directory watch handle

DosFindNotifyFirst Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to monitor changes such as create, delete, mkdir, chdir, and rename in a directory.

Parameters	Description
PSZ	Pointer to the path name string
PHDIR	The directory search handle
USHORT	The search attribute
PBYTE	Pointer to the result buffer

USHORT	The result buffer length
PUSHORT	Pointer to the number of changes required
USHORT	The request level
ULONG	The timeout value
ULONG	This is a reserved parameter and must be zero

DosFindNotifyNext Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to monitor changes such as create, delete, mkdir, chdir, and rename in a directory.

Parameters	Description
HDIR	The directory search handle
PBYTE	Pointer to the result buffer
USHORT	The result buffer length
PUSHORT	Pointer to the number of changes required
ULONG	The timeout value

DosFlagProcess Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used asynchronously to notify another process of an external event by setting a user-defined flag.

Parameters	Description
PID	The ID of the process to receive notification
USHORT	Process tree notification indicator
USHORT	The flag being set
USHORT	The arguments to be passed to the flagged process

DosFreeModule Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to terminate the reference to a dynamic link module.

Parameter	Description
HMODULE	Dynamic link module handle

DosFreeSeg Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to deallocate a memory segment.

Parameter	Description
SEL	Selector of segment to be deallocated

DosFSAttach Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to attach or detach a drive from a remote file system driver.

Parameters	Description
PSZ	Pointer to the device name string
PSZ	Pointer to the file system driver name string
PBYTE	Pointer to the data buffer
USHORT	The data buffer length
USHORT	The flag used to specify an attach or detach request
ULONG	This parameter is reserved and must be zero

DosFSCTL Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used as an extended standard interface between an application and a file system driver.

Parameters	Description
PBYTE	Pointer to the data area
USHORT	The data area length
PUSHORT	Pointer to data length returned by the file system driver
PBYTE	Pointer to the parameter list
USHORT	The parameter list length
PUSHORT	Pointer to returned parameter list length
USHORT	The file system function code
PSZ	Pointer to the file system driver name or path name string
HFILE	The file handle
USHORT	The routing request code
ULONG	This is a reserved parameter and must be zero

DosFSRamSemClear Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to clear a fast, safe RAM semaphore.

Parameter	Description
PDOSFSRSEM	Pointer to fast, safe RAM semaphore structure

DosFSRamSemRequest Failure returns an error code; otherwise, zero is returned (USHORT).

This function obtains a fast, safe RAM semaphore and sets it owned by the requesting process.

Parameters	Description
PDOSFSRSEM	Pointer to fast, safe RAM semaphore structure
LONG	Length of time in milliseconds to wait before timing out

DosGetCollate Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to obtain the collating sequence table used by the sort utility.

Parameters	Description
USHORT	Length of receiving buffer
PCOUNTRYCODE	Pointer to the country information structure
PCHAR	Pointer to the receiving buffer
PUSHORT	Pointer to the returned table length

DosGetCp Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to obtain the current process' code page.

Parameters	Description
USHORT	The receiving buffer length
PUSHORT	Pointer to the receiving buffer
PUSHORT	Pointer to the returned data length

DosGetCtryInfo Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to obtain country-dependent information.

Parameters	Description
USHORT	The receiving buffer length
PCOUNTRYCODE	Pointer to the country information structure
PCOUNTRYINFO	Pointer to the country information buffer
PUSHORT	Pointer to the returned data length

DosGetDateTime Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to obtain the current system date and time.

Parameter	Description
PDATETIME	Pointer to the date/time structure

DosGetDBCSEv Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to obtain the double-byte character set environmental vector.

Parameters	Description
USHORT	The receiving buffer length
PCOUNTRYCODE	Pointer to the country information structure
PCHAR	Pointer to the receiving buffer

DosGetEnv Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to obtain the address of the process environment strings for the current process.

Parameters	Description
PUSHORT	Pointer to the selector for the environment string
PUSHORT	Pointer to the command line offset

DosGetHugeShift Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to obtain the shift count used to determine the next selector for huge memory segments.

Parameter	Description
PUSHORT	Pointer to the shift count

DosGetInfoSeg Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to obtain the selector used to access the global data segment containing system specific information and the selector used to access the data segment containing the process specific information.

Parameters	Description
PSEL	Pointer to the global segment selector
PSEL	Pointer to the process specific segment selector

DosGetMachineMode Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to obtain the current processor mode.

Parameter	Description
PBYTE	Pointer to the mode byte

DosGetMessage Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to retrieve a message from a message file.

Parameters	Description
PCHAR	Pointer to string table
USHORT	Number of table entries
PCHAR	Pointer to message buffer
USHORT	Message buffer length
USHORT	Requested message number
PSZ	Pointer to the message file path and name string
PUSHORT	Pointer to the message length

DosGetModHandle Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to obtain the module handle for a previously loaded dynamic link library.

Parameters	Description
PSZ	Pointer to the module name string
PHMODULE	Pointer to the module handle

DosGetModName Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to determine the drive, path, and name of the module.

Parameters	Description
HMODULE	Dynamic link module handle
USHORT	Length of receiving buffer
PCHAR	Pointer to the receiving buffer

DosGetPID Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to get the ID of the current process, the current thread ID, and the parent process ID.

Parameter	Description
PPIDINFO	Pointer to the process ID structure

DosGetPPID Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to get the parent process ID.

Parameters	Description
USHORT	Process ID of process whose parent is wanted
PUSHORT	Pointer to parent process ID

DosGetProcAddr Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to get the address of a routine within a dynamic link library.

Parameters	Description
HMODULE	Dynamic link module handle
PSZ	Pointer to the routine name string
PFN	Pointer to the routine address

DosGetPrty Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to determine a process' priority.

Parameters	Description
USHORT	The indicator of whether the first thread or a specific thread priority is to be returned
PUSHORT	Pointer to the requested priority
USHORT	The process or thread ID

DosGetResource Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to get the segment selector for a resource.

Parameters	Description
HMODULE	Dynamic link module handle to get resource from
USHORT	The 16-bit resource type ID
USHORT	The 16-bit resource name ID
PSEL	Pointer to the resource selector

DosGetSeg Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to get access to a previously shared segment.

Parameter	Description
SEL	The selector used to access the segment

DosGetShrSeg Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to get access to a named segment that is previously allocated.

Parameters	Description
PSZ	Pointer to the name string
PSEL	Pointer to the selector of shared segment

DosGetVersion Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to obtain the OS/2 version number.

Parameter	Description
PUSHORT	Pointer to the OS/2 version number

DosGiveSeg Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to give access to a shared memory segment.

Parameters	Description
SEL	The caller's segment selector
PID	The receiving process' ID
PSEL	The address where the receiving process' segment selector is stored

DosHoldSignal Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to disable or enable signals for the current process.

Parameter	Description
USHORT	The enable or disable indicator

DosInsMessage Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to insert a variable text string in a message.

APPENDIX A

Parameters	Description
PCHAR	Pointer to string table
USHORT	Number of table entries
PSZ	Pointer to input message
USHORT	Input message length
PCHAR	Pointer to updated message
USHORT	Length of update buffer
PUSHORT	Pointer to the updated message length

DosKillProcess Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to terminate a process and return the termination code.

Parameters	Description
USHORT	This parameter is used to indicate whether only the process or the entire process tree will be determined
PID	ID of process or root of process tree

DosLoadModule Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to explicitly load a dynamic link library.

Parameters	Description
PSZ	Pointer to the buffer used to provide additional information if the call fails
USHORT	Length of the information buffer
PSZ	Pointer to the dynamic link library name string
PHMODULE	Pointer to the dynamic link library module handle

DosLockSeg Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to prevent a data segment allocated as discardable from being discarded while it is being used.

Parameter	Description
SEL	Selector of the segment to lock

DosMakeNmPipe Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to create a named pipe.

Parameters	Description
PSZ	Pointer to the pipe name string
PHPIPE	Pointer to the pipe handle
USHORT	This parameter is used to determine inheritance, write through, and access
USHORT	This parameter is used to determine the pipe specific mode
USHORT	The advisory outgoing buffer size
USHORT	The advisory incoming buffer size
ULONG	This parameter is used as the default timeout for DosWaitNmPipe

DosMakePipe Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to create an unnamed pipe.

Parameters	Description
PHFILE	Pointer to the pipe read handle
PHFILE	Pointer to the pipe write handle
USHORT	Size of the pipe in bytes

DosMemAvail Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to get the size of the largest available free memory block.

Parameter	Description
PULONG	Pointer to the size of available memory

DosMkDir Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to create a subdirectory.

Parameters	Description
PSZ	Pointer to the new directory name string
ULONG	This parameter is reserved and must be zero

DosMkDir2 Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to provide support for extended attributes when you create a directory.

Parameters	Description
PSZ	Pointer to the new directory name string
PEAOP	Pointer to the extended attribute structure
ULONG	This is a reserved parameter and must be zero

DosMonClose Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to close a device monitor.

Parameter	Description
HMONITOR	Device monitor handle to be closed

DosMonOpen Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to open a device monitor.

Parameters	Description
PSZ	Pointer to the device name string
PHMONITOR	Pointer to device monitor handle

DosMonRead Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to read data from a device monitor's input stream and place the data into the monitor's data area.

Parameters	Description
PBYTE	Pointer to the monitor input buffer
UCHAR	The block or run indicator
PBYTE	Pointer to the monitor's private buffer
PUSHORT	Pointer to the monitor's buffer size or length of data returned

DosMonReg Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to register an input buffer and an output buffer that will be used to monitor the device.

Parameters	Description
HMONITOR	The monitor handle returned from DosMonOpen
PBYTE	Pointer to the monitor input buffer
PBYTE	Pointer to the monitor output buffer
USHORT	This parameter is used to indicate where in the monitor chain the monitor being registered is to be placed
USHORT	Device data stream index; it is the session ID for keyboard and mouse monitors

DosMonWrite Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to write data from a monitor's private buffer to the output stream of the device monitor.

Parameters	Description
PBYTE	Pointer to the monitor's output buffer
PBYTE	Pointer to the monitor's private buffer
USHORT	Number of bytes requested to be written and the number of bytes actually written

DosMove Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to move a file from one directory to another directory.

Parameters	Description
PSZ	Pointer to the old path and file name string
PSZ	Pointer to the new path and file name string
ULONG	This parameter is reserved and must be zero

DosMuxSemWait Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to wait for two or more RAM and/or system semaphores concurrently.

Parameters	Description
PUSHORT	Pointer to semaphore index that tells which semaphore was cleared
PVOID	Pointer to the list of semaphores
LONG	This parameter is used to specify a timeout value in milliseconds

DosNewSize Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to extend the size of a file without writing any data.

Parameters	Description
HFILE	Handle of the file to be extended
ULONG	New size of the file in bytes

DosOpen Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to open a file for input or output.

Parameters	Description
PSZ	Pointer to the file's path and name string
PHFILE	Pointer to the file handle
PUSHORT	Pointer to the action taken if file is opened

ULONG	Size of the file
USHORT	The file attribute
USHORT	The action to be taken if the file exists or does not exist
USHORT	The open mode of the file
ULONG	This parameter is reserved and must be zero

DosOpen2 Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to open a file for input or output and provides additional support for extended attributes.

Parameters	Description
PSZ	Pointer to the file path and name string
PHFILE	The address for the new file's handle
PUSHORT	Pointer to the action taken if file is opened
ULONG	The size of the file
USHORT	The file attributes
USHORT	The action to be taken if the file exists or does not exist
ULONG	The open mode of the file
PEAOP	Pointer to the extended attribute structure
ULONG	This is a reserved parameter and must be zero

DosOpenQueue Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to create a queue.

Parameters	Description
PUSHORT	Pointer to the queue owner's PID
PHQUEUE	Pointer to the queue handle
PSZ	Pointer to the queue name string

DosOpenSem Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to get access to a previously created system semaphore.

Parameters	Description
PHSEM	Pointer to the semaphore handle
PSZ	Pointer to the semaphore name string

DosPeekNmPipe Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to read from a named pipe without removing data from the pipe.

Parameters	Description
HPIPE	Handle to a pipe
PBYTE	Pointer to the pipe buffer
USHORT	Buffer length
PUSHORT	Pointer to the number of bytes read
PUSHORT	Pointer to the number of bytes available
PUSHORT	Pointer to the pipe state

DosPeekQueue Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to read data from a queue without removing the data from the queue.

Parameters	Description
HQUEUE	The handle of the queue to read from
PULONG	Pointer to the request identification data
PUSHORT	Pointer to the element length
PULONG	Pointer to the receive buffer
PUSHORT	Pointer to the element indicator
UCHAR	This parameter is used to indicate whether the call should wait if the queue is empty
PBYTE	Pointer to the element priority
ULONG	Data available semaphore handle

DosPFSActivate Failure returns an error code; otherwise, zero is returned (USHORT).
 This function is used to specify which code page and font to make active for a given printer and file system.

Parameters	Description
PVOID	The spool file handle
PULONG	The address where number of bytes written is stored
PSZ	The address of printer name string
USHORT	The code page to be used
USHORT	The font to be used
USHORT	The system file number
ULONG	This parameter is reserved and must be zero

DosPFSCloseUser Failure returns an error code; otherwise, zero is returned (USHORT).
 This function is used to notify the font switcher that the spool file is closed.

Parameters	Description
PSZ	The name of the printer used
USHORT	The system file number
ULONG	This parameter is reserved and must be zero

DosPFSInit Failure returns an error code; otherwise, zero is returned (USHORT).
 This function is used to initialize the code page switching or font switching for a printer.

Parameters	Description
USHORT	Pointer list for code page and fonts
PSZ	File name for font switching and code page switching
PSZ	The printer type ID
PSZ	The name of printer used
USHORT	Maximum number of instances of font and code page
ULONG	This parameter is reserved and must be zero

DosPFSQueryAct Failure returns an error code; otherwise, zero is returned (USHORT).
 This function is used to determine which font and code page is active for a printer.

Parameters	Description
PSZ	The pointer to name of the printer used
PUSHORT	The address of active code page
PUSHORT	The address of active font

USHORT	System file number of requester
ULONG	This parameter is reserved and must be zero

DosPFSVerifyFont Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to verify that the specified font is available for the specified printer.

Parameters	Description
PSZ	The name of the printer used
PSZ	The code page to validate
USHORT	The font ID to validate
ULONG	This parameter is reserved and must be zero

DosPhysicalDisk Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to request disk partition information.

Parameters	Description
USHORT	Type of information requested
PBYTE	Pointer to the information buffer
USHORT	The information buffer length
PBYTE	Pointer to user-supplied information
USHORT	Length of user-supplied information

DosPortAccess Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used by an IOPL routine to request or release the ability to read or write to I/O ports.

Parameters	Description
USHORT	This parameter is reserved and must be zero
USHORT	This is used as the request or release indicator
USHORT	The number of the first port to be accessed
USHORT	The number of the last port to be accessed

DosPtrace Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to debug the system functions requested by a child process.

Parameter	Description
PRTRACEBUF	Pointer to the trace buffer

DosPurgeQueue Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to flush a queue.

Parameter	Description
HQUEUE	The handle of queue to be purged

DosPutMessage Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to put a message into a message file or display the message on a device.

Parameters	Description
USHORT	The handle of the message file or output device
USHORT	The length of the message buffer
PCHAR	Pointer to the message buffer

DosQAppType Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to determine the type of application in an executable file.

Parameters	Description
PSZ	Pointer to the executable file name string
PUSHORT	Pointer to the application type flags

DosQCurDir Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to determine the name of the current directory.

Parameters	Description
USHORT	The drive number
PBYTE	Pointer to the information buffer
PUSHORT	Pointer to the information length

DosQCurDisk Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to determine the default drive for the process.

Parameters	Description
PUSHORT	Pointer to the current drive number
PULONG	Pointer to drive map data area

DosQFHandState Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to determine the state of a file, which reflects the way the file was opened (its open mode).

Parameters	Description
HFILE	The handle of the file to query
PUSHORT	Pointer to the state information

DosQFileInfo Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to determine the properties of a file.

Parameters	Description
HFILE	The handle of the file to query
USHORT	The file property requested
PBYTE	Pointer to the information buffer
USHORT	The information buffer length

DosQFileMode Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to determine the current file attribute.

Parameters	Description
PSZ	Pointer to the file path and name string
PUSHORT	Pointer to the attribute information
ULONG	This parameter is reserved and must be zero

DosQFSAttach Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to query information about an attached file system driver.

Parameters	Description
PSZ	Pointer to the device name string
USHORT	The drive ordinal

USHORT	The FSAinfolevel is the level of information requested
PBYTE	Pointer to the data buffer
PUSHORT	Pointer to the data buffer length on input and the length of returned data on output
ULONG	This is a reserved parameter and must be zero

DosQFSInfo Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to obtain the file system parameters.

Parameters	Description
USHORT	The drive number of interest
USHORT	The file system data requested
PBYTE	Pointer to the request buffer
USHORT	The request buffer length

DosQHandType Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to determine what the given handle references.

Parameters	Description
HFILE	The handle to be queried
PUSHORT	Pointer to the handle class
PUSHORT	Pointer to the handle type

DosQNmPHandState Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to determine the named pipe handle state.

Parameters	Description
HPIPE	The handle of the pipe to be queried
PUSHORT	Pointer to the pipe handle state

DosQNmPipeInfo Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to obtain information about a named pipe.

Parameters	Description
HPIPE	The handle of the pipe to be queried
USHORT	The level of information requested
PBYTE	Pointer to the information buffer
USHORT	The information buffer length

DosQNmPipeSemState Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to obtain information about a named pipe attached to a system semaphore.

Parameters	Description
HSEM	The system semaphore handle
PBYTE	Pointer to the information buffer
USHORT	The information buffer length

DosQPathInfo Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to obtain information for a specific file or directory.

Parameters	Description
PSZ	Pointer to the path string
USHORT	The level of path information requested
PBYTE	Pointer to the path data buffer
USHORT	The path data buffer size
ULONG	This is a reserved parameter and must be zero

DosQSysInfo Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to obtain a copy of the system constants like the maximum path name length.

Parameters	Description
USHORT	The ordinal of system variable requested
PBYTE	Pointer to the data buffer
USHORT	The data buffer size

DosQueryQueue Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to determine the number of entries in a queue.

Parameters	Description
HQUEUE	The handle of the queue to query
PUSHORT	Pointer to the number of queue entries

DosQVerify Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to determine the state of a process' verify flag.

Parameter	Description
PUSHORT	Pointer to the verify flag

DosR2StackRealloc Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to increase the size of an IOPL routine's stack.

Parameter	Description
USHORT	The new stack size

DosRead Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to perform a synchronous read of data from a file or device.

Parameters	Description
HFILE	The file or device read handle
PVOID	Pointer to the read buffer
USHORT	The number of bytes requested
PUSHORT	Pointer to the number of bytes read

DosReadAsync Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to read data from a file or device and then continue processing while waiting for the data to be returned.

Parameters	Description
HFILE	The file or device read handle
PULONG	Pointer to the RAM semaphore used to indicate the completion of the read request
PUSHORT	Pointer to the I/O error return code
PVOID	Pointer to the read buffer
USHORT	The number of bytes to read
PUSHORT	Pointer to the number of bytes read

DosReadQueue Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to read data from a queue by the owner of the queue.

Parameters	Description
HQUEUE	The handle of queue to read from
PULONG	Pointer to the element encoding
PUSHORT	Pointer to the element length
PULONG	Pointer to the element received
USHORT	The queue element to be read
UCHAR	This parameter indicates what to do if the queue is empty
PBYTE	Pointer to the element pointer
HSEM	The handle of the semaphore associated with the queue data availability

DosReallocHuge Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to increase the size of a previously allocated huge block of memory.

Parameters	Description
USHORT	The number of 64KB segments requested
USHORT	The number of bytes in last segment
SEL	The original selector for huge memory block

DosReallocSeg Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to increase the size of a previously allocated segment.

Parameters	Description
USHORT	The new size requested in bytes
SEL	The segment selector

DosResumeThread Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to restart a suspended thread.

Parameter	Description
TID	The ID of the thread to be suspended

DosRmdir Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to delete a directory.

Parameters	Description
PSZ	Pointer to the directory name string
ULONG	This parameter is reserved and must be zero

DosScanEnv Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to scan a process' environment segment for a string.

Parameters	Description
PSZ	Pointer to the search string
PSZ FAR *	Pointer to the environment string pointer

DosSearchPath Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to search a given path for a specific file.

Parameters	Description
USHORT	This parameter is used to indicate whether the current path will be searched
PSZ	Pointer to the search path string
PSZ	Pointer to the file name string
PBYTE	Pointer to the information buffer
USHORT	The length of the returned path string

DosSelectDisk Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to change the default drive for a process.

Parameter	Description
USHORT	The default drive number

DosSelectSession Failure returns an error code; otherwise, zero is returned (USHORT).

This function allows a process to select one of its child sessions to bring to the foreground.

Parameters	Description
USHORT	The child session ID
ULONG	This parameter is reserved and must be zero

DosSemClear Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to clear a previously set semaphore.

Parameter	Description
HSEM	The handle of the semaphore to clear

DosSemRequest Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to request a semaphore.

Parameters	Description
HSEM	The handle of the semaphore to request
LONG	The time in milliseconds to wait before function times out

DosSemSet Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to set a semaphore as owned.

Parameter	Description
HSEM	The handle of the semaphore

DosSemSetWait Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to set a RAM or system semaphore and wait for the semaphore to be cleared later by some other thread or process.

Parameters

HSEM

LONG

Description

The semaphore handle to set and wait

The time in milliseconds to wait before function times out

DosSemWait Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to wait for a previously set semaphore to be cleared.

Parameters

HSEM

LONG

Description

The semaphore handle to wait for

The time in milliseconds to wait before function times out

DosSendSignal Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to send a CNTRL_C or a CNTRL_BREAK signal to a process in the subtree.

Parameters

USHORT

USHORT

Description

Process ID of root of subtree

Signal number to be sent

DosSetCp Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to change the default code pages for a process, keyboard, and display.

Parameters

USHORT

USHORT

Description

The code page identifier

This parameter is reserved and must be zero

DosSetDateTime Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to change the current system date and time.

Parameter

PDATETIME

Description

Pointer to the date and time structure

DosSetFHandState Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to establish the state for the specified file.

Parameters

HFILE

USHORT

Description

The file handle of the specified file

The new file handle state

DosSetFileInfo Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to modify the specified file properties.

Parameters

HFILE

USHORT

PBYTE

USHORT

Description

The file handle of the specified file

The file information level structure

Pointer to the file information buffer

The length of the information string

DosSetFileMode Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to change a file's attribute.

Parameters	Description
PSZ	Pointer to the file path and name string
USHORT	The new file attribute
ULONG	This parameter is reserved and must be zero

DosSetFSInfo Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to change file system information.

Parameters	Description
USHORT	The drive number of the drive to be changed
USHORT	File system information level structure
PBYTE	Pointer to the information buffer
USHORT	Length of the information string

DosSetMaxFH Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to set the maximum number of file handles for a process.

Parameter	Description
USHORT	The new file handle limit

DosSetNmPHandState Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to set the named pipe handle state.

Parameters	Description
HPIPE	The handle of the affected pipe
USHORT	The new file handle state

DosSetNmPipeSem Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to attach a named pipe to a system semaphore.

Parameters	Description
HPIPE	The handle to the named pipe
HSEM	The semaphore handle
USHORT	The key value

DosSetPathInfo Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to specify information for a file or directory.

Parameters	Description
PSZ	Pointer to the path string
USHORT	The level of information being defined
PBYTE	Pointer to the path information buffer
USHORT	The path information buffer length
USHORT	The path info flags
ULONG	This is a reserved parameter and must be zero

DosSetProcCp Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to set the process' code page.

Parameters	Description
USHORT	The code page ID
USHORT	This parameter is reserved and must be zero

DosSetPrty Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to set the priority for a process or a thread.

Parameters	Description
USHORT	Priority scope indicator
USHORT	Priority class to set
SHORT	Priority level to apply
USHORT	Process ID or thread ID

DosSetSession Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to set a child session status.

Parameters	Description
USHORT	The session ID of session being changed
PSTATUSDATA	The session status data

DosSetSigHandler Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to establish a signal handler for a process.

Parameters	Description
PFNSIGHANDLER	Pointer to the signal handler routine
PFNSIGHANDLER FAR *	Pointer to the previous signal handler
PUSHORT	Pointer to the previous action
USHORT	Request indicator type
USHORT	Signal number of interest

DosSetVec Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to establish a fault handler for a process.

Parameters	Description
USHORT	Vector number
PFN	Pointer to the exception handler
PFN	Pointer to the previous exception handler

DosSetVerify Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to set or reset the verify switch.

Parameter	Description
USHORT	New verify switch value

DosSizeSeg Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to determine the size of a segment.

Parameters	Description
SEL	The segment selector
PULONG	Pointer to size segment

APPENDIX A

DosSleep Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to suspend the execution of a thread for a given period.

Parameter	Description
ULONG	The time interval in milliseconds

DosSMRegisterDD Failure returns an error; otherwise, zero is returned (USHORT).
This function is used to register a device driver with the session manager so that the device driver will receive notification when a session switch occurs.

Parameter	Description
REGISTERDATA	The structure containing the registration information

DosStartSession Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to start a process in a separate session.

Parameters	Description
PSTARTDATA	Start session data structure containing information used to start the new session
PUSHORT	Pointer to the new session ID
PUSHORT	Pointer to the new process ID

DosStopSession Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to stop a child session.

Parameters	Description
USHORT	The stop option
USHORT	The affected session ID
ULONG	This parameter is reserved and must be zero

DosSubAlloc Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to suballocate a segment of memory that was previously initialized for suballocation.

Parameters	Description
SEL	The suballocated segment's selector
PUSHORT	Pointer to offset of the piece to be suballocated
USHORT	Length of suballocated memory

DosSubFree Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to deallocate a piece of suballocated memory.

Parameters	Description
SEL	The suballocated segment's selector
USHORT	The offset of memory to be deallocated
USHORT	Length of the deallocated memory

DosSubSet Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to initialize a memory segment for suballocation.

Parameters	Description
SEL	The suballocated segment's selector
USHORT	The suballocation indicator
USHORT	Size of memory to be suballocated

DosSuspendThread Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to suspend the execution of a thread until it is restarted.

Parameter	Description
TID	Thread ID of thread to suspend

DosTimerAsync Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to start an asynchronous timer used for timeout conditions.

Parameters	Description
ULONG	The time interval in milliseconds
HSEM	The system semaphore used to indicate the timer expiration
PHTIMER	Timer handle used to stop the timer

DosTimerStart Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to start a periodic timer.
The system semaphore will be cleared whenever the specified timer expires.

Parameters	Description
ULONG	The time interval in milliseconds
HSEM	The system semaphore used to indicate the expiration
PHTIMER	The timer handle used to stop the timer

DosTimerStop Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to stop an asynchronous timer or a periodic timer.

Parameter	Description
HTIMER	Timer handle for an asynchronous timer or periodic timer

DosTransactNmPipe Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to perform a transaction to a named pipe.

Parameters	Description
HPIPE	Named pipe handle
PBYTE	Pointer to the output buffer
USHORT	The number of bytes to write
PBYTE	Pointer to the input buffer
USHORT	The number of bytes read
PUSHORT	Pointer to bytes read

DosUnlockSeg Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to unlock a memory segment that was allocated as discardable and is not going to be referenced in a short time.

Parameter	Description
SEL	Selector of segment to unlock

DosWaitNmPipe Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to wait until an instance of a named pipe is available.

Parameters	Description
PSZ	Pointer to the pipe name string
ULONG	The maximum time to wait in milliseconds

DosWrite Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to perform a synchronous write to a file or device.

Parameters	Description
HFILE	The file handle used to write to the file
PVOID	Pointer to the output buffer
USHORT	The number of bytes requested to write
PUSHORT	The number of bytes actually written

DosWriteAsync Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to perform an asynchronous write to a file or device so the process may do other processing while it waits for the write to complete.

Parameters	Description
HFILE	The file handle used to write to the file
PULONG	Pointer to the RAM semaphore used to indicate completion of the write request
PUSHORT	Pointer to I/O error return code
PVOID	Pointer to the information buffer
USHORT	The number of bytes requested to write
PUSHORT	Pointer to the actual number of bytes written

DosWriteQueue Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used by requesting processes to put elements on a queue opened by another process.

Parameters	Description
HQUEUE	The handle of the queue to which the elements will be added
USHORT	Request encoding data
USHORT	The element length
PBYTE	Pointer to the element being added
UCHAR	Priority of element being added

GpiAssociate Returns pass or fail status (BOOL).
This function is used to associate a device context with a presentation space.

Parameters	Description
HPS	The presentation space handle
HDC	The device context handle

GpiBeginArea Returns pass or fail status (BOOL).
This function is used to indicate the start of a graphics area definition.

Parameters

HPS
ULONG

Description

The presentation space handle
This parameter is used to indicate how the area should be drawn

GpiBeginElement Returns pass or fail status (BOOL).

This function is used to indicate the start of a graphics element definition.

Parameters

HPS
LONG

Description

The presentation space handle
This is the element type, with a range of x'81000000' through x'FF000000'
Pointer to the element descriptive string

GpiBeginPath Returns pass or fail status (BOOL).

This function is used to indicate the start of a graphics path definition.

Parameters

HPS
LONG

Description

The presentation space handle
This is the path ID and must be a value of one

GpiBitBit Returns pass, correlate, or fail status (LONG).

This function is used to copy bits from a source bitmap to a destination bitmap or device.

Parameters

HPS
HPS
LONG

Description

This is the target presentation space handle
This is the source presentation space handle
This is a count of the points to be passed in the array of points
Pointer to the array of points used to do the bit blit
This parameter is used to control the mixing of the bits
This parameter is used to control the compressing of the bits if any is to be done

GpiBox Returns pass, correlate, or fail status (LONG).

This function is used to draw a box.

Parameters

HPS
LONG
PPOINTL

Description

The presentation space handle
This parameter determines how the box is to be drawn
Pointer to the point that completes the diagonal used to draw the box
The horizontal rounding control
The vertical rounding control

GpiCallSegmentMatrix Returns pass, correlate, or fail status (LONG).

This function is used to call an unchained segment from a segment chain.

Parameters

HPS
LONG

Description

The presentation space handle
The ID of segment to be called

APPENDIX A

LONG	The number of elements in the transform matrix
PMATRIXLF	Pointer to the transform matrix that is going to be applied to the called segment
LONG	This parameter is used to specify how the transform matrix is to be applied

GpiCharString Returns pass, correlate, or fail status (LONG).

This function is used to draw a character string starting at the current location.

Parameters	Description
HPS	The presentation space handle
LONG	The number of characters to draw
PCH	Pointer to the character string to be drawn

GpiCharStringAt Returns pass, correlate, or fail status (LONG).

This function is used to draw a character string starting at the location specified.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to the starting location
LONG	The number of characters to draw
PCH	Pointer to the character string to be drawn

GpiCharStringPos Returns pass, correlate, or fail status (LONG).

This function is used to control the positioning of each letter in a character string starting at the current location.

Parameters	Description
HPS	The presentation space handle
PRECTL	The background rectangle of the string
ULONG	The formatting options
LONG	The number of characters to draw
PCH	Pointer to the character string to be drawn
PLONG	Pointer to an array of increments used to position characters after the first character of the string is placed

GpiCharStringPosAt Returns pass, correlate, or fail status (LONG).

This function is used to control the positioning of each letter in a character string starting at a specified location.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to the starting location
PRECTL	Pointer to the background rectangle of the string
ULONG	The formatting options
LONG	The number of characters to draw
PCH	Pointer to the character string to be drawn
PLONG	Pointer to an array of increments used to position characters after the first of the string is placed

GpiCloseFigure Returns pass or fail status (BOOL).

This function is used to close a figure being drawn within a path definition.

Parameter	Description
HPS	The presentation space handle

GpiCloseSegment Returns pass or fail status (BOOL).
This function is used to close the definition of a graphics segment.

Parameter	Description
HPS	The presentation space handle

GpiCombineRegion Returns the complexity of the region or error status (LONG).
This function is used to combine two regions.

Parameters	Description
HPS	The presentation space handle
HRGN	The destination region handle
HRGN	The first source region handle
HRGN	The second source region handle
LONG	This parameter specifies how the regions are to be combined

GpiComment Returns pass or fail status (BOOL).
This function is used to add a comment to a graphics segment.

Parameters	Description
HPS	The presentation space handle
LONG	The length of the comment
PBYTE	Pointer to the comment string

GpiConvert Returns pass or fail status (BOOL).
This function is used to convert an array of coordinates from one coordinate space to another.

Parameters	Description
HPS	The presentation space handle
LONG	The source coordinate space
LONG	The target coordinate space
LONG	The number of points to convert
PPOINTL	Pointer to the array of (x,y) coordinates

GpiCopyMetaFile Returns a handle to a metafile or error status (HMF).
This function is used to copy the content of a loaded metafile into a metafile it creates.

Parameter	Description
HMF	The source metafile handle

GpiCorrelateChain Returns the number of correlation points or error status (LONG).
This function is used to perform a correlation on a segment chain.

Parameters	Description
HPS	The presentation space handle
LONG	The segment type on which correlation is to occur
PPOINTL	Pointer to the pick aperture center
LONG	The maximum number of hits to be returned

LONG The number of segment/tag pairs returned by each hit
 PLONG Pointer to the array to receive the segment identifiers and tags

GpiCorrelateFrom Returns the number of correlation points or error status (LONG).
 This function is used to request correlation on a part of a segment chain.

Parameters	Description
HPS	The presentation space handle
LONG	The first segment to be correlated
LONG	The last segment to be correlated
LONG	The type of segments on which correlation is to be performed
PPOINTL	Pointer to the pick aperture center
LONG	The maximum number of hits to be returned
LONG	The number of segment/tag pairs returned by each hit
PLONG	Pointer to the array to receive the segment identifiers and tags

GpiCorrelateSegment Returns the number of correlation points or error status (LONG).
 This function is used to request correlation on a specific segment.

Parameters	Description
HPS	The presentation space handle
LONG	The ID of the segment to be correlated
LONG	The type of segments on which correlation is to be performed
PPOINTL	Pointer to the pick aperture center
LONG	The maximum number of hits to be returned
LONG	The number of segment/tag pairs returned by each hit
PLONG	Pointer to the array to receive the segment identifiers and tags

GpiCreateBitmap Returns the handle to the bitmap or error status (HBITMAP).
 This function is used to create (start the definition of) a bitmap.

Parameters	Description
HPS	The presentation space handle
PBITMAPINFOHEADER	Pointer to the bitmap information header structure
ULONG	The options used by the device
PBYTE	Pointer to the bitmap buffer if the data is provided when the bitmap is created
PBITMAPINFO	Pointer to the bitmap information table

GpiCreateLogColorTable Returns pass or fail status (BOOL).
 This function is used to define the entries of a color table for an application.

Parameters	Description
HPS	The presentation space handle
ULONG	The options used to create the color table
LONG	The format of entries in the table
LONG	The starting index
LONG	The number of elements in table
PLONG	Pointer to the color table definition data

GpiCreateLogFont Returns the match or error status (LONG).

This function is used to define a logical font for an application.

Parameters	Description
HPS	The presentation space handle
PSTR8	Pointer to the logical font name
LONG	The local ID
PFATTRS	Pointer to the attribute structure used to create the logical font

GpiCreatePS Returns the handle of a graphics presentation space (HPS).

This function is used to create a graphics presentation space.

Parameters	Description
HAB	The anchor block handle
HDC	The device context handle
PSIZEL	Pointer to the presentation page size
ULONG	The options used to create the presentation space

GpiCreateRegion Returns the handle to a region or error status (HRGN).

This function is used to create a region.

Parameters	Description
HPS	The presentation space handle
LONG	The number of rectangles used to define the region
PRECTL	Pointer to the array of rectangles used to define the region

GpiDeleteBitmap Returns pass or fail status (BOOL).

This function is used to delete a bitmap.

Parameter	Description
HBITMAP	The handle of bitmap to be deleted

GpiDeleteElement Returns pass or fail status (BOOL).

This function is used to delete an element from a graphics segment.

Parameter	Description
HPS	The presentation space handle

GpiDeleteElementRange Returns pass or fail status (BOOL).

This function is used to delete two or more elements from a graphics segment.

Parameters	Description
HPS	The presentation space handle
LONG	The number of the first element to be deleted
LONG	The number of the last element to be deleted

GpiDeleteElementsBetweenLabels Returns pass or fail status (BOOL).

This function is used to delete the elements between two labels from a graphics segment.

Parameters	Description
HPS	The presentation space handle
LONG	The starting label
LONG	The ending label

GpiDeleteMetaFile Returns pass or fail status (BOOL).

This function is used to delete program access to a memory metafile.

Parameter	Description
HMF	The handle of the metafile to be deleted

GpiDeleteSegment Returns pass or fail status (BOOL).

This function is used to delete a graphics segment.

Parameters	Description
HPS	The presentation space handle
LONG	The segment ID

GpiDeleteSegments Returns pass or fail status (BOOL).

This function is used to delete a range of segments.

Parameters	Description
HPS	The presentation space handle
LONG	The first segment ID in the range
LONG	The last segment ID in the range

GpiDeleteSetId Returns pass or fail status (BOOL).

This function is used to delete a logical font or bitmap tag.

Parameters	Description
HPS	The presentation space handle
LONG	The local ID to be deleted

GpiDestroyPS Returns pass or fail status (BOOL).

This function is used to delete the specified presentation space.

Parameter	Description
HPS	The presentation space handle

GpiDestroyRegion Returns pass or fail status (BOOL).

This function is used to delete the specified region.

Parameters	Description
HPS	The presentation space handle
HRGN	The handle of region to be deleted

GpiDrawChain Returns pass or fail status (BOOL).

This function is used to draw a graphics segment chain.

Parameter	Description
HPS	The presentation space handle

GpiDrawDynamics Returns pass or fail status (BOOL).

This function is used to draw segments that have the dynamic attribute specified.

Parameter	Description
HPS	The presentation space handle

GpiDrawFrom Returns pass or fail status (BOOL).

This function is used to draw a part of a segment chain.

Parameters	Description
HPS	The presentation space handle
LONG	The ID of first segment to be drawn
LONG	The ID of last segment to be drawn

GpiDrawSegment Returns pass or fail status (BOOL).

This function is used to draw an unchained segment.

Parameters	Description
HPS	The presentation space handle
LONG	The ID of segment to be drawn

GpiElement Returns pass, correlate, or error status (LONG).

This function is used to add an element to a graphics segment.

Parameters	Description
HPS	The presentation space handle
LONG	The element type with a value between x'81000000' and x'FF000000'
PSZ	Pointer to the element description string
LONG	The length of the element
PBYTE	Pointer to the buffer that defines the element

GpiEndArea Returns pass, correlate, or error status (LONG).

This function is used to close an area definition.

Parameter	Description
HPS	The presentation space handle

GpiEndElement Returns pass or fail status (BOOL).

This function is used to close an element definition.

Parameter	Description
HPS	The presentation space handle

GpiEndPath Returns pass or fail status (BOOL).

This function is used to close a path definition.

Parameter	Description
HPS	The presentation space handle

GpiEqualRegion Returns equality or fail status (LONG).

This function is used to determine whether two regions have the same shape.

Parameters	Description
HPS	The presentation space handle
HRGN	The handle of first region
HRGN	The handle of second region

GpiErase Returns pass or fail status (BOOL).

This function is used to clear a presentation space.

Parameter	Description
HPS	The presentation space handle

GpiErrorSegmentData Returns the error offset within a graphics segment or error status (LONG).

This function is used to obtain information about the last error that occurred while a segment was drawn.

Parameters	Description
HPS	The presentation space handle
PLONG	Pointer to the segment ID in which the error occurred
PLONG	Pointer to the error context indicator

GpiExcludeClipRectangle Returns the clipping complexity or error status (LONG).

This function is used to exclude a rectangle from the clipping region.

Parameters	Description
HPS	The presentation space handle
PRECTL	Pointer to the rectangle to be excluded

GpiFillPath Returns pass, correlate, or fail status (LONG).

This function is used to draw the interior of a defined path.

Parameters	Description
HPS	The presentation space handle
LONG	Path ID; must be 1
LONG	The fill option to use

GpiFullArc Returns pass, correlate, or fail status (LONG).

This function is used to draw a circle or an ellipse.

Parameters	Description
HPS	The presentation space handle
LONG	This parameter is used to indicate how the interior should be drawn
FIXED	This parameter is used to set the size

GpiGetData Returns the number of bytes retrieved or error status (LONG).

This function is used to copy data from a graphics segment.

Parameters	Description
HPS	The presentation space handle
LONG	The segment ID
PLONG	Pointer to the segment offset where copying will start

LONG	The coordinate type required
LONG	The length of copy buffer
PBYTE	Pointer to the copy buffer

GpiImage Returns pass, correlate, or fail status (LONG).

This function is used to obtain a rectangular image from a user-supplied buffer.

Parameters	Description
HPS	The presentation space handle
LONG	The image format; must be zero
PSIZEL	Pointer to image area
LONG	The image buffer length
PBYTE	Pointer to the buffer of image data

GpiIntersectClipRectangle Returns the clipping complexity or error status (LONG).

This function is used to set a new clipping region.

Parameters	Description
HPS	The presentation space handle
PRECTL	Pointer to the rectangle used to set new region

GpiLabel Returns pass or fail status (BOOL).

This function is used to place a label in a graphics segment.

Parameters	Description
HPS	The presentation space handle
LONG	The label to be added

GpiLine Returns pass, correlate, or fail status (LONG).

This function is used to draw a line between the current position and the end point passed as a parameter.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to the end point

GpiLoadBitmap Returns the handle to a bitmap or error status (HBITMAP).

This function is used to load a bitmap from a resource file.

Parameters	Description
HPS	The presentation space handle
HMODULE	The handle of the dynamic link library containing the bitmap
USHORT	The bitmap ID
LONG	The width of the bitmap in pels
LONG	The height of the bitmap in pels

GpiLoadFonts Returns pass or fail status (BOOL).

This function is used to load a font from a resource file.

Parameters	Description
HAB	The anchor block handle
PSZ	Pointer to the resource file name string

GpiLoadMetaFile Returns the handle to a metafile or error status (HMF).

This function is used to load the contents of a resource file that is a metafile.

Parameters	Description
HAB	The anchor block handle
PSZ	Pointer to the resource file name string

GpiMarker Returns pass, correlate, or fail status (LONG).

This function is used to draw a marker with its center at the point specified.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to the marker center location

GpiModifyPath Returns pass or fail status (BOOL).

This function is used to change a previously defined path.

Parameters	Description
HPS	The presentation space handle
LONG	The path ID must be 1
LONG	The requested modification

GpiMove Returns pass or fail status (BOOL).

This function is used to change the current position.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to the new location

GpiOffsetClipRegion Returns the clipping complexity or error status (LONG).

This function is used to change the offset of the clipping region.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to the clipping region offset

GpiOffsetElementPointer Returns pass or fail status (BOOL).

This function is used to change the offset of the element pointer within the segment.

Parameters	Description
HPS	The presentation space handle
LONG	The offset to be added to the element pointer

GpiOffsetRegion Returns pass or fail status (BOOL).

This function is used to move a region.

Parameters	Description
HPS	The presentation space handle
HRGN	The handle of the region to be moved
PPOINTL	Pointer to the offset to be added to the region boundary

GpiOpenSegment Returns pass or fail status (BOOL).

This function is used to create a graphics segment or reopen a previously created segment.

Parameters

HPS
LONG

Description

The presentation space handle
The ID of segment to be opened

GpiOutlinePath Returns pass, correlate, or fail status (LONG).

This function is used to draw the outline of a path.

Parameters

HPS
LONG
LONG

Description

The presentation space handle
The path identifier; it must be 1
This parameter is reserved and must be zero

GpiPaintRegion Returns pass, correlate, or fail status (LONG).

This function is used to paint a region in a presentation space.

Parameters

HPS
HRGN

Description

The presentation space handle
The handle of the region to be painted

GpiPartialArc Returns pass, correlate, or fail status (LONG).

This function is used to draw part of an arc.

Parameters

HPS
PPOINTL
FIXED
FIXED
FIXED

Description

The presentation space handle
Pointer to the arc central point
The size of the arc
The start of the angle in degrees
The sweep of the angle in degrees

GpiPlayMetaFile Returns pass, correlate, or fail status (LONG).

This function is used to display the contents of a metafile that was previously loaded by GpiLoadMetafile.

Parameters

HPS
HMF
LONG
PLONG
PLONG
LONG
PSZ

Description

The presentation space handle
The handle to the metafile
The number of elements in the optional array
Pointer to the array of options used to play the metafile
Pointer to the count of renumbered segments
The length of the descriptive record buffer
Pointer to the descriptive record buffer

GpiPointArc Returns pass, correlate, or fail status (LONG).

This function is used to draw a three-point arc starting from the current location.

Parameters

HPS
PPOINTL

Description

The presentation space handle
Pointer to the intermediate and end points

GpiPolyFillet Returns pass, correlate, or fail status (LONG).

This function is used to draw multiple arcs.

Parameters

HPS
LONG
PPOINTL

Description

The presentation space handle
The number of arcs to be drawn
Pointer to the array of points used to draw the arcs

GpiPolyFilletSharp Returns pass, correlate, or fail status (LONG).

This function is used to draw a fillet from a series of connected lines.

Parameters

HPS
LONG
PPOINTL
PPFIXED

Description

The presentation space handle
The number of points provided
Pointer to the array of points
Pointer to the array of sharpness values

GpiPolyLine Returns pass, correlate, or fail status (LONG).

This function is used to draw multiple connected lines from a single instruction.

Parameters

HPS
LONG
PPOINTL

Description

The presentation space handle
The number of points provided
Pointer to the array of end points

GpiPolyMarker Returns pass, correlate, or fail status (LONG).

This function is used to draw multiple markers located at the points supplied from a single instruction.

Parameters

HPS
LONG
PPOINTL

Description

The presentation space handle
The number of markers to be drawn
Pointer to the array of points used to draw the markers

GpiPolySpline Returns pass, correlate, or fail status (LONG).

This function is used to draw multiple curves from a single instruction.

Parameters

HPS
LONG
PPOINTL

Description

The presentation space handle
The number of control points used to draw the curves
Pointer to the array of control points

GpiPop Returns pass or fail status (BOOL).

This function is used to restore the previously saved attributes.

Parameters

HPS
LONG

Description

The presentation space handle
The number of attributes to be restored

GpiPtInRegion Returns inside or outside region or error status (LONG).

This function is used to determine whether a point lies within the region.

Parameters	Description
HPS	The presentation space handle
HRGN	The handle of the region
PPOINTL	Pointer to the point to be checked

GpiPtVisible Returns visibility or error status (LONG).

This function is used to determine the visibility of a point.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to the point to be checked

GpiPutData Returns pass, correlate, or fail status (LONG).

This function is used to copy data to a segment from a buffer.

Parameters	Description
HPS	The presentation space handle
LONG	The coordinate type used
PLONG	Pointer to the buffer length
PBYTE	Pointer to the data buffer

GpiQueryArcParams Returns pass or fail status (BOOL).

This function is used to determine the current value of the arc parameters.

Parameters	Description
HPS	The presentation space handle
PARCPARAMS	Pointer to the arc parameter structure

GpiQueryAttrMode Returns the attribute mode or error status (LONG).

This function is used to determine the current attribute mode.

Parameter	Description
HPS	The presentation space handle

GpiQueryAttrs Returns default mask or error status (LONG).

This function is used to determine the current presentation space attributes.

Parameters	Description
HPS	The presentation space handle
LONG	The primitive type to be queried
ULONG	The attributes to be queried
PBUNDLE	Pointer to the attribute buffer

GpiQueryBackColor Returns background color or error status (LONG).

This function is used to determine the background color.

Parameter	Description
HPS	The presentation space handle

GpiQueryBackMix Returns background mix mode or error status (LONG).

This function is used to determine the background mix.

Parameter	Description
HPS	The presentation space handle

GpiQueryBitmapBits Returns the number of scan lines or error status (LONG).
This function is used to transfer bitmap data to application storage.

Parameters	Description
HPS	The presentation space handle
LONG	The starting scan line
LONG	The number of scan lines to transfer
PBYTE	Pointer to the data buffer
PBITMAPINFO	Pointer to the bitmap information structure

GpiQueryBitmapDimension Returns pass or fail status (BOOL).
This function is used to determine the size of a bitmap.

Parameters	Description
HBITMAP	The handle of the bitmap to be queried
PSIZEL	Pointer to the bitmap size

GpiQueryBitmapHandle Returns bitmap handle or error status (HBITMAP).
This function is used to determine the handle of the bitmap currently associated with the local ID.

Parameters	Description
HPS	The presentation space handle
LONG	The local ID

GpiQueryBitmapParameters Returns pass or fail status (BOOL).
This function is used to obtain the bitmap information.

Parameters	Description
HBITMAP	The handle of the bitmap
PBITMAPINFOHEADER	Pointer to the bitmap information header structure

GpiQueryBoundaryData Returns pass or fail status (BOOL).
This function is used to obtain the presentation space boundary.

Parameters	Description
HPS	The presentation space handle
PRECTL	Pointer to the boundary data

GpiQueryCharAngle Returns pass or fail status (BOOL).
This function is used to determine the current character angle.

Parameters	Description
HPS	The presentation space handle
PGRADIENTL	Pointer to the baseline angle

GpiQueryCharBox Returns pass or fail status (BOOL).
This function is used to obtain the size of the character box.

Parameters	Description
HPS	The presentation space handle
PSIZEF	Pointer to the character box size

GpiQueryCharDirection Returns the direction or fail status (LONG).
This function is used to obtain the current value of the character direction attribute.

Parameter	Description
HPS	The presentation space handle

GpiQueryCharMode Returns the character mode or error status (LONG).
This function is used to determine the current character mode.

Parameter	Description
HPS	The presentation space handle

GpiQueryCharSet Returns the local ID or error status (LONG).
This function is used to determine the current character local ID.

Parameter	Description
HPS	The presentation space handle

GpiQueryCharShear Returns pass or fail status (BOOL).
This function is used to obtain the current character shear.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to the character shear

GpiQueryCharStringPos Returns pass or fail status (BOOL).
This function is used to determine the position of each character in a string, starting at the current location.

Parameters	Description
HPS	The presentation space handle
ULONG	The options used in the query
LONG	The string length
PCH	Pointer to the character string
PLONG	Pointer to the array of x increment values
PPOINTL	Pointer to the array of character locations

GpiQueryCharStringPosAt Returns pass or fail status (BOOL).
This function is used to determine the position of each character in a string, starting at the position specified.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to string starting location
ULONG	The options used in the query
LONG	The string length
PCH	Pointer to the character string

APPENDIX A

PLONG Pointer to the array of x increment values
 PPOINTL Pointer to the array of character locations

GpiQueryClipBox Returns the clipping complexity or error status (LONG).
 This function is used to determine the current clip box size.

Parameters	Description
HPS	The presentation space handle
PRECTL	Pointer to the clip box size

GpiQueryClipRegion Returns a region handle or error status (HRGN).
 This function is used to determine the current clipping region.

Parameter	Description
HPS	The presentation space handle

GpiQueryColor Returns a color or error status (LONG).
 This function is used to determine the current foreground color.

Parameter	Description
HPS	The presentation space handle

GpiQueryColorData Returns pass or fail status (BOOL).
 This function is used to obtain the color table information.

Parameters	Description
HPS	The presentation space handle
LONG	The number of elements requested
PLONG	Pointer to the information array

GpiQueryColorIndex Returns the color index or error status (LONG).
 This function is used to determine the color index of the specified color.

Parameters	Description
HPS	The presentation space handle
ULONG	The query options
LONG	The RGB color

GpiQueryCp Returns a code page or error status (USHORT).
 This function is used to determine the current code page.

Parameter	Description
HPS	The presentation space handle

GpiQueryCurrentPosition Returns pass or fail status (BOOL).
 This function is used to determine the current presentation space location.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to the current position

GpiQueryDefArcParams Returns pass or fail status (BOOL).
 This function is used to obtain the value of the default arc parameters.

Parameters	Description
HPS	The presentation space handle
PARCPARAMS	The address to which to return the arc parameters

GpiQueryDefAttr Returns pass or fail status (BOOL).

This function is used to obtain the default values for the requested primitive.

Parameters	Description
HPS	The presentation space handle
LONG	The primitive type
ULONG	The attribute mask used to determine the values to return
PBUNDLE	Pointer to the attribute structure to be updated

GpiQueryDefaultViewMatrix Returns pass or fail status (BOOL).

This function is used to obtain the default viewing transform matrix.

Parameters	Description
HPS	The presentation space handle
LONG	The number of elements requested
PMATRIXLF	Pointer to the transform matrix

GpiQueryDefCharBox Returns pass or fail status (BOOL).

This function is used to determine the default character box size.

Parameters	Description
HPS	The presentation space handle
PSIZEL	Pointer to the default character box size

GpiQueryDefTag Returns pass or fail status (BOOL).

This function is used to return the value of the default tag identifier set by GpiSetDefTag.

Parameters	Description
HPS	The presentation space handle
PLONG	The address to which the default tag identifier is returned

GpiQueryDefViewingLimits Returns pass or fail status (BOOL).

This function is used to obtain the default viewing limits set by GpiSetDefViewingLimits.

Parameters	Description
HPS	The presentation space handle
PRECTL	Pointer to where the default viewing limits are returned

GpiQueryDevice Returns a device context handle or error status (HDC).

This function is used to obtain the handle of the currently associated device context.

Parameter	Description
HPS	The presentation space handle

GpiQueryDeviceBitmapFormats Returns pass or fail status (BOOL).

This function is used to determine the bitmap formats supported by a device.

Parameters	Description
HPS	The presentation space handle
LONG	The number of elements requested
PLONG	Pointer to the format information

GpiQueryDrawControl Returns current setting of the drawing control or error status (LONG).

This function is used to determine the current drawing control value.

Parameters	Description
HPS	The presentation space handle
LONG	The control whose value is to be returned

GpiQueryDrawingMode Returns the drawing mode or error status (LONG).

This function is used to determine the current drawing mode.

Parameter	Description
HPS	The presentation space handle

GpiQueryEditMode Returns the current edit mode or error status (LONG).

This function is used to determine the current edit mode.

Parameter	Description
HPS	The presentation space handle

GpiQueryElement Returns the element length or error status (LONG).

This function is used to determine the content of the requested element.

Parameters	Description
HPS	The presentation space handle
LONG	The starting byte offset within the element
LONG	The length of the request buffer
PBYTE	Pointer to the request buffer

GpiQueryElementPointer Returns the element pointer or error status (LONG).

This function is used to determine the current element pointer.

Parameter	Description
HPS	The presentation space handle

GpiQueryElementType Returns the length of element type or error status (LONG).

This function is used to obtain information on an element.

Parameters	Description
HPS	The presentation space handle
PLONG	Pointer to the element type
LONG	The length of the information buffer
PSZ	Pointer to the information buffer

GpiQueryFontFileDescriptions Returns the number of fonts that do not have details or error status (LONG).

This function is used to obtain information from a font file.

Parameters	Description
HAB	The anchor block handle
PSZ	Pointer to the file name string
PLONG	Pointer to request count
PFFDESCS	Pointer to the array of font file descriptors

GpiQueryFontMetrics Returns pass or fail status (BOOL).
This function is used to determine the characteristics of a font.

Parameters	Description
HPS	The presentation space handle
LONG	The length of the information buffer
PFONTMETRICS	Pointer to the information buffer

GpiQueryFonts Returns the count of fonts not returned or error status (LONG).
This function is used to determine the number of fonts loaded in the system.

Parameters	Description
HPS	The presentation space handle
ULONG	This parameter is used to query private or public fonts
PSZ	Pointer to the facename of fonts
PLONG	The count of fonts requested
LONG	The length of the information structure
PFONTMETRICS	Pointer to the information buffer

GpiQueryGraphicsField Returns pass or fail status (BOOL).
This function is used to determine the graphics field rectangle.

Parameters	Description
HPS	The presentation space handle
PRECTL	Pointer to the graphics field rectangle

GpiQueryInitialSegmentAttrs Returns the current segment attribute status or error status (LONG).

This function is used to determine the value of the initial segment attributes.

Parameters	Description
HPS	The presentation space handle
LONG	The attribute to be queried

GpiQueryKerningPairs Returns the number of kerning pairs or error status (LONG).
This function is used to determine the number of kerning pairs supported by the current font.

Parameters	Description
HPS	The presentation space handle
LONG	The number of elements requested
PKERNINGPAIRS	Pointer to the array of kerning pairs

GpiQueryLineEnd Returns the line end or error status (LONG).
This function is used to determine the current line end setting.

APPENDIX A

Parameter	Description
HPS	The presentation space handle

GpiQueryLineJoin Returns the line join type or error status (LONG).
This function is used to determine how crossing lines will be joined.

Parameter	Description
HPS	The presentation space handle

GpiQueryLineType Returns the line type or error status (LONG).
This function is used to determine the current line type setting.

Parameter	Description
HPS	The presentation space handle

GpiQueryLineWidth Returns the line width or error status (LONG).
This function is used to determine the current line width.

Parameter	Description
HPS	The presentation space handle

GpiQueryLineWidthGeom Returns the geometric line width or error status (LONG).
This function is used to determine the geometric line width.

Parameter	Description
HPS	The presentation space handle

GpiQueryLogColorTable Returns the number of elements in logical color table or error status (LONG).

This function is used to obtain the current logical color table.

Parameters	Description
HPS	The presentation space handle
ULONG	The options used to process the request
LONG	The starting index for which data is to be returned
LONG	The count of elements requested
PLONG	Pointer to the color table array

GpiQueryMarker Returns the current marker symbol or error status (LONG).
This function is used to determine the current marker symbol.

Parameter	Description
HPS	The presentation space handle

GpiQueryMarkerBox Returns pass or fail status (BOOL).
This function is used to determine the size of the marker.

Parameters	Description
HPS	The presentation space handle
PSIZEF	Pointer to the size of marker box

GpiQueryMarkerSet Returns the marker set local ID or error status (LONG).
This function is used to determine the local ID for the current marker.

Parameter	Description
HPS	The presentation space handle

GpiQueryMetaFileBits Returns pass or fail status (BOOL).

This function is used to copy information from a metafile to an application buffer.

Parameters	Description
HMF	The handle to a memory metafile
LONG	The starting offset byte
LONG	The information buffer length
PBYTE	Pointer to the information buffer

GpiQueryMetaFileLength Returns the metafile length or error status (LONG).

This function is used to determine the length of a metafile.

Parameter	Description
HMF	The metafile handle

GpiQueryMix Returns the mix mode or error status (LONG).

This function is used to obtain the foreground color mix.

Parameter	Description
HPS	The presentation space handle

GpiQueryModelTransformMatrix Returns pass or fail status (BOOL).

This function is used to obtain the current model transform matrix.

Parameters	Description
HPS	The presentation space handle
LONG	The number of elements requested
PMATRIXLF	Pointer to the transform matrix structure

GpiQueryNearestColor Returns the closest color or error status (LONG).

This function is used to obtain the nearest color available to the specified color.

Parameters	Description
HPS	The presentation space handle
ULONG	The options used to do the query
LONG	The requested color

GpiQueryNumberSetIds Returns the number of set IDs in use or error status (LONG).

This function is used to obtain the number of set IDs in use.

Parameter	Description
HPS	The presentation space handle

GpiQueryPageViewport Returns pass or fail status (BOOL).

This function is used to determine the current page viewport.

Parameters	Description
HPS	The presentation space handle
PRECTL	Pointer to the page viewport

GpiQueryPattern Returns the current pattern symbol or error status (LONG).
This function is used to obtain the current pattern symbol.

Parameter	Description
HPS	The presentation space handle

GpiQueryPatternRefPoint Returns pass or fail status (BOOL).
This function is used to obtain the current pattern reference point.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to the pattern reference point

GpiQueryPatternSet Returns the current pattern set ID or error status (LONG).
This function is used to obtain the current pattern set local ID.

Parameter	Description
HPS	The presentation space handle

GpiQueryPel Returns the pel color index or error status (LONG).
This function is used to determine the pel color index at a specified location.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to the location in world coordinates

GpiQueryPickAperturePosition Returns pass or fail status (BOOL).
This function is used to determine the correlation pick aperture location.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to the pick aperture location

GpiQueryPickApertureSize Returns pass or fail status (BOOL).
This function is used to determine the size of the correlation pick aperture size.

Parameters	Description
HPS	The presentation space handle
PSIZEL	Pointer to the pick aperture size

GpiQueryPS Returns the presentation space options or error status (LONG).
This function is used to obtain the presentation page size for the presentation space.

Parameters	Description
HPS	The presentation space handle
PSIZEL	Pointer to the presentation page size

GpiQueryRealColors Returns the number of elements returned or error status (LONG).
This function is used to determine the RGB color values of the colors available.

Parameters	Description
HPS	The presentation space handle
ULONG	The options used for the query

LONG	The ordinal number of the first color requested
LONG	The information buffer length
PLONG	Pointer to the information buffer

GpiQueryRegionBox Returns the region complexity or error status (LONG).

This function is used to obtain the region rectangle.

Parameters	Description
HPS	The presentation space handle
HRGN	The region handle
PRECTL	Pointer to the region rectangle

GpiQueryRegionRects Returns pass or fail status (BOOL).

The function is used to determine the rectangles that define the specified region.

Parameters	Description
HPS	The presentation space handle
HRGN	The region handle
PRECTL	Pointer to the region rectangle
PRGNRECT	Processing control structure
PRECTL	Pointer to the array of rectangles

GpiQueryRGBColor Returns the closest RGB color or error status (LONG).

This function is used to determine the RGB value associated with an index.

Parameters	Description
HPS	The presentation space handle
ULONG	The options used for the query
LONG	The color index

GpiQuerySegmentAttrs Returns the current attribute value or error status (LONG).

This function is used to determine the current segment attributes.

Parameters	Description
HPS	The presentation space handle
LONG	The segment ID
LONG	The attribute to be queried

GpiQuerySegmentNames Returns the number of IDs returned or error status (LONG).

This function is used to determine the name of segments that exist within a specified range.

Parameters	Description
HPS	The presentation space handle
LONG	The first segment of the range
LONG	The last segment of the range
LONG	The maximum number of segments requested
PLONG	Pointer to the array of segment names

GpiQuerySegmentPriority Returns the segment ID or error status (LONG).

This function is used to determine the ID of the segment that comes before or follows the specified segment.

Parameters	Description
HPS	The presentation space handle
LONG	The specified segment ID
LONG	Segment request higher or lower option

GpiQuerySegmentTransformMatrix Returns pass or fail status (BOOL).
This function is used to obtain the current segment transform matrix.

Parameters	Description
HPS	The presentation space handle
LONG	The number of elements requested
PMATRIXLF	Pointer to the transform matrix

GpiQuerySetIds Returns pass or fail status (BOOL).
This function is used to obtain information for loaded fonts or bitmaps.

Parameters	Description
HPS	The presentation space handle
LONG	The number of local IDs in use
PLONG	Request type, font, or bitmap
PSTR8	Pointer to array of font names
PLONG	Pointer to array of local identifiers

GpiQueryStopDraw Returns the stop draw status or error status (LONG).
This function is used to determine whether the stop draw condition exists.

Parameter	Description
HPS	The presentation space handle

GpiQueryTag Returns pass or fail status (BOOL).
This function is used to determine the current value of the tag ID.

Parameters	Description
HPS	The presentation space handle
PLONG	Pointer to the tag identifier

GpiQueryTextBox Returns pass or fail status (BOOL).
This function is used to determine the size of the box bounding a text string.

Parameters	Description
HPS	The presentation space handle
LONG	The string length
PCH	Pointer to the character string
LONG	The number of points requested
PPOINTL	Pointer to array of points

GpiQueryViewingLimits Returns pass or fail status (BOOL).
This function is used to determine the viewing rectangle size.

Parameters	Description
HPS	The presentation space handle
PRECTL	Pointer to the viewing rectangle

GpiQueryViewingTransformMatrix Returns pass or fail status (BOOL).

This function is used to determine the current viewing transform matrix.

Parameters	Description
HPS	The presentation space handle
LONG	The number of elements requested
PMATRIXLF	Pointer to the transform matrix

GpiQueryWidthTable Returns pass or fail status (BOOL).

This function is used to determine the font width information for the selected font.

Parameters	Description
HPS	The presentation space handle
LONG	The code point of first character
LONG	The number of elements requested
PLONG	Pointer of the array of width values

GpiRealizeColorTable Returns pass or fail status (BOOL).

This function is used to change the color table.

Parameter	Description
HPS	The presentation space handle

GpiRectInRegion Returns the region location status or error status (LONG).

This function is used to determine whether a part of a rectangle is in the specified region.

Parameters	Description
HPS	The presentation space handle
HRGN	The region handle
PRECTL	Pointer to the rectangle

GpiRectVisible Returns the visibility status or error status (LONG).

This function is used to determine whether any part of a rectangle is visible.

Parameters	Description
HPS	The presentation space handle
PRECTL	Pointer to the rectangle

GpiRemoveDynamics Returns pass or fail status (BOOL).

This function is used to remove parts of a picture drawn from dynamic segments.

Parameters	Description
HPS	The presentation space handle
LONG	The first segment to remove
LONG	The last segment to remove

GpiResetBoundaryData Returns pass or fail status (BOOL).

This function is used to reset boundary data.

Parameter	Description
HPS	The presentation space handle

GpiResetPS Returns pass or fail status (BOOL).

This function is used to reset a presentation space as if it were just created.

Parameters	Description
HPS	The presentation space handle
ULONG	The reset option

GpiRestorePS Returns pass or fail status (BOOL).

This function is used to return the presentation space to its state when the GpiSavePS was issued.

Parameters	Description
HPS	The presentation space handle
LONG	The presentation space ID to restore

GpiSaveMetaFile Returns pass or fail status (BOOL).

This function is used to write a metafile to disk.

Parameters	Description
HMF	The metafile handle
PSZ	Pointer to the path and file name string

GpiSavePS Returns the presentation space ID or error status (LONG).

This function is used to save a presentation space.

Parameter	Description
HPS	The presentation space handle

GpiSetArcParams Returns pass or fail status (BOOL).

This function is used to change the arc parameters.

Parameters	Description
HPS	The presentation space handle
PARCPARAMS	Pointer to the arc parameter structure

GpiSetAttrMode Returns pass or fail status (BOOL).

This function is used to change the attribute mode.

Parameters	Description
HPS	The presentation space handle
LONG	This parameter is used to specify the attribute mode

GpiSetAttrs Returns pass or fail status (BOOL).

This function is used to change the attributes for a primitive.

Parameters	Description
HPS	The presentation space handle
LONG	The primitive type
ULONG	The attributes to change
ULONG	The attribute defaults mask
PBUNDLE	The value of the attribute to change

GpiSetBackColor Returns pass or fail status (BOOL).

This function is used to change the background color.

Parameters	Description
HPS	The presentation space handle
LONG	The background color

GpiSetBackMix Returns pass or fail status (BOOL).

This function is used to change the background mix.

Parameters	Description
HPS	The presentation space handle
LONG	The background mix mode

GpiSetBitmap Returns the bitmap handle status or error status (HBITMAP).

This function is used to select a bitmap into a presentation space.

Parameters	Description
HPS	The presentation space handle
HBITMAP	Handle of the bitmap to be set

GpiSetBitmapBits Returns the number of scan lines set or error status (LONG).

This function is used to change the bits in a bitmap using buffered data.

Parameters	Description
HPS	The presentation space handle
LONG	The line number of the bitmap to change
LONG	The number of scan lines to change
PBYTE	Pointer to bitmap data buffer
PBITMAPINFO	Pointer to bitmap information structure

GpiSetBitmapDimension Returns pass or fail status (BOOL).

This function is used to change the bitmap dimensions.

Parameters	Description
HBITMAP	The bitmap handle
PSIZEL	Pointer to the width and height of bitmap

GpiSetBitmapId Returns pass or fail status (BOOL).

This function is used to tag a bitmap with a local ID.

Parameters	Description
HPS	The presentation space handle
HBITMAP	The handle of the bitmap
LONG	The local ID to use

GpiSetCharAngle Returns pass or fail status (BOOL).

This function is used to change the character angle.

Parameters	Description
HPS	The presentation space handle
PGRAIDENTL	Pointer to the baseline angle

APPENDIX A

GpiSetCharBox Returns pass or fail status (BOOL).

This function is used to change the character box size.

Parameters	Description
HPS	The presentation space handle
PSIZEF	Pointer to the character box size

GpiSetCharMode Returns pass or fail status (BOOL).

This function is used to change the character mode.

Parameters	Description
HPS	The presentation space handle
LONG	The character mode to use

GpiSetCharSet Returns pass or fail status (BOOL).

This function is used to make a logical font the current font.

Parameters	Description
HPS	The presentation space handle
LONG	The local character set ID identifier

GpiSetCharShear Returns pass or fail status (BOOL).

This function is used to change the character shear.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to the character shear

GpiSetClipPath Returns pass or fail status (BOOL).

This function is used to change the clip path.

Parameters	Description
HPS	The presentation space handle
LONG	This parameter is used to specify what to do with the current clipping option
LONG	The clipping option to use

GpiSetClipRegion Returns the clipping complexity or error status (LONG).

This function is used to change the clipping region.

Parameters	Description
HPS	The presentation space handle
HRGN	The region handle
PHRGN	Pointer to the old region handle

GpiSetColor Returns pass or fail status (BOOL).

This function is used to change the foreground color.

Parameters	Description
HPS	The presentation space handle
LONG	The new color to use

GpiSetCp Returns pass or fail status (BOOL).

This function is used to change the code page.

Parameters	Description
HPS	The presentation space handle
USHORT	The code page to use

GpiSetCurrentPosition Returns pass or fail status (BOOL).

This function is used to change the current position, as is GpiMove.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to the new location

GpiSetDefArcParams Returns pass or fail status (BOOL).

This function is used to change the default arc parameter values.

Parameters	Description
HPS	The presentation space handle
PARCPARAMS	Pointer to the arc parameters

GpiSetDefAttr Returns pass or fail status (BOOL).

This function is used to change the values of the default attributes for the primitive requested.

Parameters	Description
HPS	The presentation space handle
LONG	The primitive whose attribute is to be changed
ULONG	The attribute mask used to establish which attribute is to be changed
PBUNDLE	Pointer to the attribute structure

GpiSetDefaultViewMatrix Returns pass or fail status (BOOL).

This function is used to set a new default viewing matrix.

Parameters	Description
HPS	The presentation space handle
LONG	The number of elements provided
PMATRIXLF	Pointer to the new viewing transform matrix
LONG	The transform options to use

GpiSetDefTag Returns pass or fail status (BOOL).

This function is used to change the default tag.

Parameters	Description
HPS	The presentation space handle
LONG	The default tag identifier

GpiSetDefViewingLimits Returns pass or fail status (BOOL).

This function is used to change the default viewing limits.

Parameters	Description
HPS	The presentation space handle
PRECTL	Pointer to the new default viewing limits

APPENDIX A

GpiSetDrawControl Returns pass or fail status (BOOL).

This function is used to change the drawing control.

Parameters	Description
HPS	The presentation space handle
LONG	The drawing control to set
LONG	The requested state of the control

GpiSetDrawingMode Returns pass or fail status (BOOL).

This function is used to change the drawing mode.

Parameters	Description
HPS	The presentation space handle
LONG	The drawing mode to use

GpiSetEditMode Returns pass or fail status (BOOL).

This function is used to change the graphics segment edit mode.

Parameters	Description
HPS	The presentation space handle
LONG	The new edit mode

GpiSetElementPointer Returns pass or fail status (BOOL).

This function is used to change the segment element pointer.

Parameters	Description
HPS	The presentation space handle
LONG	The required element

GpiSetElementPointerAtLabel Returns pass or fail status (BOOL).

This function is used to set an element pointer to point to a label in a graphics segment.

Parameters	Description
HPS	The presentation space handle
LONG	The element label

GpiSetGraphicsField Returns pass or fail status (BOOL).

This function is used to change the graphics field.

Parameters	Description
HPS	The presentation space handle
PRECTL	The new graphics field

GpiSetInitialSegmentAttrs Returns pass or fail status (BOOL).

This function is used to establish the initial attributes for a segment.

Parameters	Description
HPS	The presentation space handle
LONG	The segment attribute to change
LONG	This parameter is used to enable or disable the attributes

GpiSetLineEnd Returns pass or fail status (BOOL).

This function is used to change the appearance of a line ending.

Parameters

HPS

LONG

Description

The presentation space handle

The new line ending

GpiSetLineJoin Returns pass or fail status (BOOL).

This function is used to change the appearance of how lines will be joined.

Parameters

HPS

LONG

Description

The presentation space handle

The new line join setting

GpiSetLineType Returns pass or fail status (BOOL).

This function is used to change the line type.

Parameters

HPS

LONG

Description

The presentation space handle

The new line type to use

GpiSetLineWidth Returns pass or fail status (BOOL).

This function is used to change the line width.

Parameters

HPS

FIXED

Description

The presentation space handle

The new width to use

GpiSetLineWidthGeom Returns pass or fail status (BOOL).

This function is used to change the geometric line width.

Parameters

HPS

LONG

Description

The presentation space handle

The new geometric line width

GpiSetMarker Returns pass or fail status (BOOL).

This function is used to change the current marker symbol.

Parameters

HPS

LONG

Description

The presentation space handle

The new marker symbol value

GpiSetMarkerBox Returns pass or fail status (BOOL).

This function is used to change the marker box size.

Parameters

HPS

PSIZEF

Description

The presentation space handle

Pointer to the marker box size

GpiSetMarkerSet Returns pass or fail status (BOOL).

This function is used to tag a marker font with a local ID.

Parameters

HPS

LONG

Description

The presentation space handle

The local ID to use identifier

APPENDIX A

GpiSetMetaFileBits Returns pass or fail status (BOOL).

This function is used to copy metafile data to a memory buffer.

Parameters	Description
HMF	The metafile memory handle
LONG	The metafile offset
LONG	The information buffer length
PBYTE	Pointer to the information buffer

GpiSetMix Returns pass or fail status (BOOL).

This function is used to change the foreground mix.

Parameters	Description
HPS	The presentation space handle
LONG	The new mix mode

GpiSetModelTransformMatrix Returns pass or fail status (BOOL).

This function is used to change the model transform matrix.

Parameters	Description
HPS	The presentation space handle
LONG	The number of elements to change
PMATRIXLF	Pointer to the model transform matrix
LONG	The transform options to use

GpiSetPageViewport Returns pass or fail status (BOOL).

This function is used to change the page viewport.

Parameters	Description
HPS	The presentation space handle
PRECTL	Pointer to the new page viewport

GpiSetPattern Returns pass or fail status (BOOL).

This function is used to change a pattern symbol.

Parameters	Description
HPS	The presentation space handle
LONG	The new pattern symbol

GpiSetPatternRefPoint Returns pass or fail status (BOOL).

This function is used to change the pattern reference point.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to the pattern reference point

GpiSetPatternSet Returns pass or fail status (BOOL).

This function is used to change the current pattern set to the referenced local ID.

Parameters	Description
HPS	The presentation space handle
LONG	The local ID

GpiSetPel Returns pass, correlate, or fail status (LONG).

This function is used to change a pel at the referenced location.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to the pel location

GpiSetPickAperturePosition Returns pass or fail status (BOOL).

This function is used to change the correlation pick aperture position.

Parameters	Description
HPS	The presentation space handle
PPOINTL	Pointer to the center of the pick aperture

GpiSetPickApertureSize Returns pass or fail status (BOOL).

This function is used to change the size of the correlation pick aperture.

Parameters	Description
HPS	The presentation space handle
LONG	The setting option
PSIZEL	Pointer to the pick aperture size

GpiSetPS Returns pass or fail status (BOOL).

This function is used to change the presentation space size, units, and format.

Parameters	Description
HPS	The presentation space handle
PSIZEL	Pointer to the presentation space size
ULONG	The presentation space options to change

GpiSetRegion Returns pass or fail status (BOOL).

This function is used to change the region.

Parameters	Description
HPS	The presentation space handle
HRGN	The region handle
LONG	The count of rectangles
PRECTL	Pointer to the array of rectangles

GpiSetSegmentAttrs Returns pass or fail status (BOOL).

This function is used to change the segment attributes.

Parameters	Description
HPS	The presentation space handle
LONG	The segment ID
LONG	The segment attribute to change
LONG	The attribute value

GpiSetSegmentPriority Returns pass or fail status (BOOL).

This function is used to change the priority of a segment in reference to another segment.

Parameters	Description
HPS	The presentation space handle
LONG	The segment ID
LONG	The reference segment ID
LONG	This parameter is used to move the segment higher or lower than the referenced segment

GpiSetSegmentTransformMatrix Returns pass or fail status (BOOL).

This function is used to change the segment transform matrix.

Parameters	Description
HPS	The presentation space handle
LONG	The segment ID
LONG	The number of elements to change
PMATRIXLF	Pointer to the transformation matrix
LONG	Transform options

GpiSetStopDraw Returns pass or fail status (BOOL).

This function is used to change the stop draw condition.

Parameters	Description
HPS	The presentation space handle
LONG	The means of setting or clearing the stop draw condition

GpiSetTag Returns pass or fail status (BOOL).

This function is used to establish a tag in a segment used for correlation.

Parameters	Description
HPS	The presentation space handle
LONG	The tag ID

GpiSetViewingLimits Returns pass or fail status (BOOL).

This function is used to change the viewing limits.

Parameters	Description
HPS	The presentation space handle
PRECTL	Pointer to the viewing rectangle

GpiSetViewingTransformMatrix Returns pass or fail status (BOOL).

This function is used to change the viewing transform matrix.

Parameters	Description
HPS	The presentation space handle
LONG	The segment ID
LONG	The number of elements to change
PMATRIXLF	Pointer to the transformation matrix
LONG	Transform options

GpiStrokePath Returns pass, correlate, or fail status (LONG).

This function is used to fill a defined path.

Parameters	Description
HPS	The presentation space handle
LONG	Path ID to fill
ULONG	The stroke option to use

GpiUnloadFonts Returns pass or fail status (BOOL).
This function is used to unload fonts loaded by GpiLoadFonts.

Parameters	Description
HAB	The anchor block handle
PSZ	Pointer to the font path and file name string

GpiUnrealizeColorTable Returns pass or fail status (BOOL).
This function is used to unload a color table.

Parameter	Description
HPS	The presentation space handle

GpiWCBitBlT Returns pass, correlate, or fail status (LONG).
This function is used to copy a bitmap to another bitmap.

Parameters	Description
HPS	The presentation space handle
HBITMAP	The source bitmap handle
LONG	The number of points provided
PPOINTL	Pointer to array of points
LONG	How the bits are to be mixed
ULONG	How compression will be handled

KbdCharIn Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to get a character and its scan code from the keyboard.

Parameters	Description
PKBDKEYINFO	Pointer to the buffer in which the input data will be placed
USHORT	Indicator to specify whether to wait for input if none is currently available
HKBD	Handle of the logical keyboard

KbdClose Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to close a logical keyboard.

Parameter	Description
HKBD	Handle of the logical keyboard to be closed

KbdDeRegister Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to deregister a keyboard subsystem previously registered during a session.

Parameter	Description
VOID	No parameters required

KbdFlushBuffer Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to clear the keystroke buffer.

Parameter	Description
HKBD	Handle of the logical keyboard

KbdFreeFocus Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to undo the logical-to-physical keyboard connection previously established.

Parameter	Description
HKBD	Handle of the logical keyboard

KbdGetCp Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to obtain the code page currently in use.

Parameters	Description
ULONG	Reserved (must be 0)
PUSHORT	Pointer to the code page ID
HKBD	Handle of the logical keyboard

KbdGetFocus Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to establish a logical-to-physical keyboard connection.

Parameters	Description
USHORT	Indicator to specify whether to wait if the physical keyboard is currently connected to another logical keyboard
HKBD	Handle of the logical keyboard

KbdGetHWId Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to obtain the keyboard's hardware generated identification value.

Parameters	Description
PKBDHWID	Pointer to the structure to receive the information
HKBD	The keyboard handle

KbdGetStatus Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to obtain the current state of the keyboard.

Parameters	Description
PKBDINFO	Pointer to the structure into which the keyboard data will be placed
HKBD	Handle of the logical keyboard

KbdOpen Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to create a logical keyboard.

Parameter	Description
PHKBD	Pointer to the logical keyboard handle

KbdPeek Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to retrieve keyboard data from the buffer without removing the data from the buffer.

Parameters
 PKBDKEYINFO
 HKBD

Description
 Pointer to the buffer in which the input data will be placed
 Handle of the logical keyboard

KbdRegister Failure returns an error code; otherwise, zero is returned (USHORT).
 This function is used to register a keyboard subsystem within a session.

Parameters
 PSZ
 PSZ
 ULONG

Description
 Name of the dynamic link module for the keyboard subsystem
 Name of the entry point in the dynamic link module that
 gains control when a registered keyboard function is called
 Mask of bits defining the keyboard functions to be registered

KbdSetCp Failure returns an error code; otherwise, zero is returned (USHORT).
 This function is used to set the code page to use in a keyboard translation.

Parameters
 USHORT
 USHORT
 HKBD

Description
 Reserved (must be 0)
 ID of the code page to be used
 Handle of the logical keyboard

KbdSetCustXt Failure returns an error code; otherwise, zero is returned (USHORT).
 This function is used to set a custom code page.

Parameters
 PUSHORT
 HKBD

Description
 Pointer to the translation table that defines a custom code
 page
 Handle of the logical keyboard

KbdSetFgnd Failure returns an error code; otherwise, zero is returned (USHORT).
 This function is used to boost the priority of the threads in the current process that owns the keyboard.

Parameter
 VOID

Description
 No parameters required

KbdSetStatus Failure returns an error code; otherwise, zero is returned (USHORT).
 This function is used to set the characteristics of the logical keyboard.

Parameters
 PKBDINFO
 HKBD

Description
 Pointer to the structure containing the keyboard characteristics
 Handle of the logical keyboard

KbdStringIn Failure returns an error code; otherwise, zero is returned (USHORT).
 This function is used to read a character string from the keyboard.

Parameters
 PCH
 PSTRINGINBUF

Description
 Pointer to the buffer into which the character string will be
 placed
 Pointer to the structure that will contain the length of the input
 buffer and in which the length of the input string will
 be stored

APPENDIX A

USHORT	Indicator to specify whether to wait if no input is available
HKBD	Handle of the logical keyboard

KbdSynch Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to synchronize access from a keyboard system to the keyboard device driver.

Parameter	Description
USHORT	Indicator to specify whether to wait for access

KbdXlate Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to translate a scan code and shift state information into an ASCII code.

Parameters	Description
PKBDTRANS	Pointer to a translation record structure
HKBD	Handle of the logical keyboard

MouClose Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to close the mouse device for the current session.

Parameter	Description
HMOU	Handle of the mouse device

MouDeRegister Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to deregister a mouse subsystem.

Parameter	Description
VOID	No parameters required

MouDrawPtr Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to notify the mouse device driver to draw the pointer image.

Parameter	Description
HMOU	Handle of the mouse device

MouFlushQue Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to flush the mouse event queue and the monitor chain data for the current session.

Parameter	Description
HMOU	Handle of the mouse device

MouGetDevStatus Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to obtain the characteristics of the installed mouse device driver.

Parameters	Description
PUSHORT	Pointer to the storage area into which the status flags will be placed
HMOU	Handle of the mouse device

MouGetEventMask Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to obtain the current value of the mouse event queue mask.

Parameters	Description
PUSHORT	Pointer to the storage area into which the mouse event queue mask will be placed
HMOU	Handle of the mouse device

MouGetNumButtons Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to obtain the numbers of buttons supported on the installed mouse device driver.

Parameters	Description
PUSHORT	Pointer to the storage area into which the number of supported buttons will be placed
HMOU	Handle of the mouse device

MouGetNumMickeys Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to obtain the number of mickeys per centimeter for the installed mouse device driver.

Parameters	Description
PUSHORT	Pointer to the storage area into which the number of physical mouse motion units will be placed
HMOU	Handle of the mouse device

MouGetNumQueEI Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to obtain the current status of the mouse device driver event queue.

Parameters	Description
PMOQUEINFO	Pointer to the structure into which the queue status will be placed
HMOU	Handle of the mouse device

MouGetPtrPos Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to obtain the current row and column coordinate position of the mouse pointer.

Parameters	Description
PPTRLOC	Pointer to the structure into which the position coordinates will be placed
HMOU	Handle of the mouse device

MouGetPtrShape Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to obtain the mouse pointer shape for the current session.

Parameters	Description
PBYTE	Pointer to the buffer in which the pointer bit image will be stored
PPTRSHAPE	Pointer to the structure in which the definition of the pointer shape will be stored
HMOU	Handle of the mouse device

MouGetScaleFact Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to obtain the scaling factors for the mouse device.

Parameters	Description
PSCALEFACT	Pointer to the structure in which will be stored the scaling factors
HMOU	Handle of the mouse device

MouInitReal Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to initialize the DOS mode mouse device driver.

Parameter	Description
PSZ	Pointer to the name of the DOS mode mouse device driver

MouOpen Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to open the mouse device driver.

Parameters	Description
PSZ	Pointer to the name of the mouse device driver
HMOU	Handle of the mouse device

MouReadEventQue Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to obtain an event from the mouse event queue.

Parameters	Description
PMOUEVENTINFO	Pointer to the structure in which the mouse event record will be stored
PUSHORT	Pointer to an indicator describing the action to be taken if the event queue is empty
HMOU	Handle of the mouse device

MouRegister Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to register a mouse subsystem.

Parameters	Description
PSZ	Pointer to the dynamic link module name of the mouse subsystem
PSZ	Pointer to the name of the entry point in the dynamic link module that receives control when any of the registered functions is called
ULONG	Mask of bits identifying the mouse functions being registered

MouRemovePtr Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to define an area in which the mouse pointer image is not to be drawn.

Parameters	Description
PNOPTRECT	Pointer to the data structure defining the rectangle of exclusion
HMOU	Handle of the mouse device

MouSetDevStatus Failure returns an error code; otherwise, zero is returned (USHORT).
This function is used to set the mouse device driver status flags.

Parameters	Description
PUSHORT	Pointer to the desired status flag settings
HMOU	Handle of the mouse device

MouSetEventMask Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to define a new event mask for the mouse device driver.

Parameters	Description
PUSHORT	Pointer to the new mouse event mask
HMOU	Handle of the mouse device

MouSetPtrPos Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to set the position of the mouse pointer.

Parameters	Description
PPTRLOC	Pointer to the structure containing the new mouse pointer coordinates
HMOU	Handle of the mouse device

MouSetPtrShape Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to define the shape and size of the mouse pointer.

Parameters	Description
PBYTE	Pointer to the buffer containing the bit image of the new mouse pointer
PPTRSHAPE	Pointer to the structure containing the data required by the mouse device driver to use the new mouse pointer definition
HMOU	Handle of the mouse device

MouSetScaleFact Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to define new scaling factors for the mouse device driver.

Parameters	Description
PSCALEFACT	Pointer to the structure containing the new scaling factors
HMOU	Handle of the mouse device

MouSynch Failure returns an error code; otherwise, zero is returned (USHORT).

This function is used to define synchronous access for a mouse subsystem to the mouse device driver.

Parameter	Description
USHORT	Indicator specifying whether to wait for access if the mouse device driver is busy

PrfAddProgram Failure returns null; otherwise, handle (HPROGRAM) is returned.

This function is used to add a program entry to a group in the program list.

Parameters	Description
HINI	The initialization file handle
PPROGDETAILS	Detail information for program to be added
HPROGRAM	The handle of the group to which the program will be added

PrfChangeProgram Returns pass or fail status (BOOL).

This function is used to change the contents of a program entry in the program list.

Parameters	Description
HINI	The initialization file handle
HPROGRAM	The handle of the program whose information is to be changed
PPROGDETAILS	Detail information for the program change

PrfCloseProfile Returns pass or fail status (BOOL).

This function is used to terminate access to a profile file.

Parameter	Description
HINI	The initialization file handle

PrfDestroyGroup Returns pass or fail status (BOOL).

This function is used to remove a group definition from the program list.

Parameters	Description
HINI	The initialization file handle
HPROGRAM	The handle of the group to be removed

PrfOpenProfile Failure returns null; otherwise, an initialization file handle (HINI) is returned.

This function is used to gain access to a profile file.

Parameters	Description
HAB	The anchor block handle
PSZ	Pointer to the profile file name string

PrfQueryProfile Returns pass or fail status (BOOL).

This function is used to obtain a description of the current user profile or system profile.

Parameters	Description
HAB	The anchor block handle
PPRFPFILE	Pointer to the profile names structure

PrfQueryProfileData Returns pass or fail status (BOOL).

This function is used to obtain a binary data string from the requested profile file.

Parameters	Description
HINI	The initialization file handle
PSZ	Pointer to the application name string
PSZ	Pointer to the key name string
PVOID	The buffer address where the returned data is placed
PULONG	Pointer to the data buffer size

PrfQueryProfileSize Returns pass or fail status (BOOL).

This function is used to obtain the size in bytes of the value of a specified key for a requested application in the profile.

Parameters	Description
HINI	The initialization file handle
PSZ	Pointer to the application name string
PSZ	Pointer to the key name string
PULONG	Pointer to where to receive the key length

PrfQueryProfileString Returns the length of a profile string (ULONG).

This function is used to obtain the length of a requested profile string.

Parameters	Description
HINI	The initialization file handle
PSZ	Pointer to the application name string
PSZ	Pointer to the key name string
PSZ	The address where the default string will be received
PVOID	The buffer address where the requested string will be received
ULONG	The buffer length

PrfQueryProgramCategory Returns the program type (PROG_CATEGORY).

This function is used to obtain the program type of the requested program.

Parameters	Description
HINI	The initialization file handle
PSZ	Pointer to the executable file name string

PrfQueryProgramHandle Returns a program handle (ULONG).

This function is used to obtain a handle for the requested executable file.

Parameters	Description
HINI	The initialization file handle
PSZ	Pointer to the executable file name string
PHPROGARRAY	Pointer to array to receive program handles
ULONG	The size of handles array
PULONG	The address at which to store the handles returned count

PrfQuerySystemProfile Returns the system profile handle (HINI).

This function is used to obtain the handle to the system profile.

Parameter	Description
HAB	The anchor block handle

PrfQueryUserProfile Returns the user profile handle (HINI).

This function is used to obtain the handle to a user profile.

Parameter	Description
HAB	The anchor block handle

PrfRemoveProgram Returns pass or fail status (BOOL).

This function is used to remove the definition of a program from the program list.

Parameters	Description
HINI	The initialization file handle
HPROGRAM	The handle of program being removed

APPENDIX A

PrfReset Returns pass or fail status (BOOL).

This function is used to define which files are to be used as the user and system profiles.

Parameters	Description
HAB	The anchor block handle
PPRFPFILE	Pointer to the profile name structures

PrfWriteProfileData Returns pass or fail status (BOOL).

This function is used to write a string of binary data to the requested profile.

Parameters	Description
HINI	The initialization file handle
PSZ	Pointer to the application name string
PSZ	Pointer to the key name string
PVOID	The address of the value data for the key name
ULONG	The size of the value data

PrfWriteProfileString Returns pass or fail status (BOOL).

This function is used to write a string of character data into the requested profile.

Parameters	Description
HINI	The initialization file handle
PSZ	Pointer to the application name string
PSZ	Pointer to the key name string
PSZ	Pointer to the text string

SplQmAbort Returns pass or fail status (BOOL).

This function is used to abort the generation of spool files.

Parameter	Description
HSPL	Handle of the spooler

SplQmClose Returns pass or fail status (BOOL).

This function is used to close the Spooler Queue Manager.

Parameter	Description
HSPL	Handle of the spooler

SplQmEndDoc Returns the job ID (USHORT).

This function is used to end a print job.

Parameter	Description
HSPL	Handle of the spooler

SplQmOpen Returns the handle to the spooler (HSPL).

This function is used to open the Spooler Queue Manager.

Parameters	Description
PSZ	Pointer to a token that identifies spooler information held in the initialization file
LONG	Length of the structure containing the open parameters
PQMOPENDATA	Pointer to the structure containing spooler open parameters

SplQmStartDoc Returns pass or fail status (BOOL).

This function is used to start a print job.

Parameters	Description
HSPL	Handle of the spooler
PSZ	Pointer to the name of the document to be printed

SplQmWrite Returns pass or fail status (BOOL).

This function is used to write a buffer to the spool file.

Parameters	Description
HSPL	Handle of the spooler
LONG	Length of the buffer
PBYTE	Pointer to the buffer of data to be written

SplQmWriteFile Returns pass or fail status (BOOL).

This function is used to write a file to the print spool.

Parameters	Description
HSPL	The spool handle
PSZ	Pointer to the file name string

SplQpInstall Returns pass or fail status (BOOL).

This function is used to install the queue processor.

Parameter	Description
HWND	Window handle

SplQpQueryDt Returns pass or fail status (BOOL).

This function is used to obtain a list of the data types supported by the queue processor.

Parameters	Description
PLONG	Pointer to the maximum number of data types that can be returned
PSZ	Pointer to an array in which the data types supported will be returned

VioAssociate Returns an error/noerror code (USHORT).

This function is used to associate a previously created VIO presentation space with a previously opened device context.

Parameters	Description
HDC	Device context handle to be associated
HVPS	VIO presentation space handle to be associated

VioCreateLogFont Returns an error/noerror code (USHORT).

This function is used to create a logical font to be used in a VIO window.

Parameters	Description
PFATTRS	Font attribute structure
LONG	Identifier of the font
PSTR8	Logical font name
HVPS	Handle to a VIO presentation space

VioCreatePS Returns an error/noerror code (USHORT).

This function is used to create a VIO presentation space.

Parameters	Description
PHVPS	The VIO presentation space handle returned
SHORT	Depth of the presentation space in character cells
SHORT	Width of the presentation space in character cells
SHORT	Format of the presentation space
SHORT	Number of attribute bytes per character
HVPS	Reserved (must be zero)

VioDeleteSetId Returns an error/noerror code (USHORT).

This function is used to delete a logical font.

Parameters	Description
LONG	Identifier of the font
HVPS	The VIO presentation space handle

VioDeRegister Returns an error/noerror code (USHORT).

This function removes a registered video subsystem.

Parameter	Description
VOID	No parameters required

VioDestroyPS Returns an error/noerror code (USHORT).

This function is used to destroy a previously created VIO presentation space.

Parameter	Description
HVPS	The VIO presentation space handle to be destroyed

VioEndPopUp Returns an error/noerror code (USHORT).

This call will end a VIO popup.

Parameter	Description
HVIO	Video handle (must be 0)

VioGetAnsi Returns an error/noerror code (USHORT).

This function is used to determine whether the ANSI state is on or off.

Parameters	Description
PUSHORT	The ANSI On-Off indicator returned
HVIO	Video handle

VioGetBuf Returns an error/noerror code (USHORT).

This function is used to give the address of the logical video buffer.

Parameters	Description
PULONG	Pointer to the logical video buffer
PUSHORT	Length of the logical video buffer
HVIO	Video handle

VioGetConfig Returns an error/noerror code (USHORT).

This function is used to return the current video configuration.

Parameters	Description
USHORT	Reserved (must be 0)
PVIOCONFIGINFO	Pointer to a configuration data structure
HVIO	Video handle

VioGetCp Returns an error/noerror code (USHORT).

This function is used to return the current code page information.

Parameters	Description
USHORT	Reserved (must be 0)
PUSHORT	Current code page ID returned
HVIO	Video handle

VioGetCurPos Returns an error/noerror code (USHORT).

This function is used to query the current cursor position.

Parameters	Description
PUSHORT	Cursor row returned
PUSHORT	Cursor column returned
HVIO	Video handle

VioGetCurType Returns an error/noerror code (USHORT).

This function is used to query the current cursor type.

Parameters	Description
PVIOCURLSORINFO	Pointer to the cursor information structure
HVIO	Video handle

VioGetDeviceCellSize Returns an error/noerror code (USHORT).

This function is used to get the current cell size for the device.

Parameters	Description
PSHORT	Current cell size height (in pels)
PSHORT	Current cell size width (in pels)
HVPS	Handle to a VIO presentation space

VioGetFont Returns an error/noerror code (USHORT).

This function is used to return a font table for the specified size.

Parameters	Description
PVIOFONTINFO	Pointer to the font request structure
HVIO	Video handle

VioGetMode Returns an error/noerror code (USHORT).

This function is used to return data about the current display mode.

Parameters	Description
PVIOMODEINFO	Pointer to a mode information structure
HVIO	Video handle

VioGetOrg Returns an error/noerror code (USHORT).

This function returns the origin of the VIO presentation space.

Parameters	Description
PSHORT	The origin row
PSHORT	The origin column
HVPS	The VIO presentation space handle

VioGetPhysBuf Returns an error/noerror code (USHORT).

This function is used to return the address of the physical display buffer.

Parameters	Description
PVIOPHYSBUF	Pointer to the physical buffer structure
USHORT	Reserved (must be 0)

VioGetState Returns an error/noerror code (USHORT).

This function is used to return the current video palette, overscan color, or intensity switch.

Parameters	Description
PVOID	Pointer to the request block
HVIO	Video handle

VioGlobalReg Returns an error/noerror code (USHORT).

This function is used to enable a subsystem to receive notification at the completion of VIO calls issued by all applications running in full-screen sessions.

Parameters	Description
PSZ	Pointer to a DLL module name string
PSZ	Pointer to the entry point name string
ULONG	Function bit mask one
ULONG	Function bit mask two

VioModeUndo Returns an error/noerror code (USHORT).

This function is used to enable a thread to cancel a VioModeWait set by another thread.

Parameters	Description
USHORT	Indicates if the thread wants ownership of the VioModeWait indicator
USHORT	Tells whether the thread that issued the VioModeWait should be terminated
USHORT	Reserved (must be 0)

VioModeWait Returns an error/noerror code (USHORT).

This function is used to notify a thread that it should restore its video settings after another thread has returned from a VioPopUp or hard error popup.

Parameters	Description
USHORT	Request type (only 0 is used)
PUSHORT	Notify type returned
USHORT	Reserved (must be 0)

VioPopUp Returns an error/noerror code (USHORT).

This function is used to display a popup screen.

Parameters	Description
PUSHORT	Option flags
HVIO	Video handle

VioPrtSc Returns an error/noerror code (USHORT).

This function is used to print the contents of the current screen.

Parameter	Description
HVIO	Video handle

VioPrtScToggle Returns an error/noerror code (USHORT).

This function toggles the print screen state on or off.

Parameter	Description
HVIO	Video handle

VioQueryFonts Returns an error/noerror code (USHORT).

This function is used to return information about the specified font face name.

Parameters	Description
PLONG	Number of fonts for which information is not returned
PFONTMETRICS	Pointer to the font metrics structure
LONG	The length of the metrics records
PLONG	The number of fonts for which information is returned
PSZ	The font face name
ULONG	The means of deciding whether to enumerate public or private fonts
HVPS	Handle to a VIO presentation space

VioQuerySetIds Returns an error/noerror code (USHORT).

This function is used to get the IDs of all loaded logical fonts.

Parameters	Description
PLONG	Area to which the IDs are returned
PSTR8	Array of font names that are returned
PLONG	Font object types
LONG	The number of objects to be reviewed
HVPS	Handle to a VIO presentation space

VioReadCellStr Returns an error/noerror code (USHORT).

This function is used to read a set of characters-attribute pairs starting at the specified cell.

Parameters	Description
PCH	Pointer to the buffer where string will be sent
PUSHORT	Length of the string buffer
USHORT	The row at which the pairs begin to be read
USHORT	The column at which the pairs begin to be read
HVIO	Video handle

VioReadCharStr Returns an error/noerror code (USHORT).

This function is used to read a string starting at the current cursor location.

Parameters	Description
PCH	Pointer to buffer where string will be sent
PUSHORT	Length of the string buffer
USHORT	The row at which the string begins to be read
USHORT	The column at which the string begins to be read
HVIO	Video handle

VioRegister Returns an error/noerror code (USHORT).

This function is used to register a video subsystem.

Parameters	Description
PSZ	The module where the subsystem resides
PSZ	The entry point of the subsystem
ULONG	Function bit mask one
ULONG	Function bit mask two

VioSavRedrawUndo Returns an error/noerror code (USHORT).

This function is used to enable a thread to cancel a VioSavRedrawWait set by another thread.

Parameters	Description
USHORT	The indicator of whether the thread wants ownership of the indicator
USHORT	The indicator of whether the thread that issued the VioSavRedrawWait should be terminated
USHORT	Video handle

VioSavRedrawWait Returns an error/noerror code (USHORT).

This function is used to notify an application when it must save or redraw its file.

Parameters	Description
USHORT	The indicator of whether save or redraw is requested
PUSHORT	The means of determining what operation will be performed
USHORT	Video handle

VioScrLock Returns an error/noerror code (USHORT).

This function is used to lock the screen.

Parameters	Description
USHORT	The means of determining whether the thread requesting screen I/O should block
PUCHAR	The indicator of whether the lock was successful
HVIO	Video handle

VioScrollDn Returns an error/noerror code (USHORT).

This function is used to scroll the display down.

Parameters	Description
USHORT	The top of the scroll area
USHORT	The left column of the scroll area
USHORT	The bottom of the scroll area
USHORT	The right of the scroll area

USHORT	The number of lines to scroll
PBYTE	The bytes to fill the newly emptied cell
HVIO	Video handle

VioScrollLf Returns an error/noerror code (USHORT).

This function is used to scroll the display left.

Parameters	Description
USHORT	The top of the scroll area
USHORT	The left column of the scroll area
USHORT	The bottom of the scroll area
USHORT	The right of the scroll area
USHORT	The number of lines to scroll
PBYTE	The bytes to fill the newly emptied cell
HVIO	Video handle

VioScrollRt Returns an error/noerror code (USHORT).

This function is used to scroll the display right.

Parameters	Description
USHORT	The top of the scroll area
USHORT	The left column of the scroll area
USHORT	The bottom of the scroll area
USHORT	The right of the scroll area
USHORT	The number of lines to scroll
PBYTE	The bytes to fill the newly emptied cell
HVIO	Video handle

VioScrollUp Returns an error/noerror code (USHORT).

This function is used to scroll the display up.

Parameters	Description
USHORT	The top of the scroll area
USHORT	The left column of the scroll area
USHORT	The bottom of the scroll area
USHORT	The right of the scroll area
USHORT	The number of lines to scroll
PBYTE	The bytes to fill the newly emptied cell
HVIO	Video handle

VioScrUnLock Returns an error/noerror code (USHORT).

This function is used to unlock a previously locked screen.

Parameter	Description
HVIO	Video handle

VioSetAnsi Returns an error/noerror code (USHORT).

This function is used to set the ANSI state on or off.

Parameters	Description
USHORT	The indicator of whether ANSI should be set on or off
HVIO	Video handle

VioSetCp Returns an error/noerror code (USHORT).

This function is used to set the current code page.

Parameters	Description
USHORT	Reserved (must be 0)
USHORT	The code page to be set
HVIO	Video handle

VioSetCurPos Returns an error/noerror code (USHORT).

This function is used to set the current cursor position.

Parameters	Description
USHORT	Row to be set
USHORT	Column to be set
HVIO	Video handle

VioSetCurType Returns an error/noerror code (USHORT).

This function is used to set the current cursor type.

Parameters	Description
PVIOCURLSORINFO	Pointer to the cursor information structure
HVIO	Video handle

VioSetDeviceCellSize Returns an error/noerror code (USHORT).

This function is used to set the cell size of the video device.

Parameters	Description
SHORT	Cell size height (in pels)
SHORT	Cell size width (in pels)
HVPS	Handle to a VIO presentation space

VioSetFont Returns an error/noerror code (USHORT).

This function is used to set a display font.

Parameters	Description
PVIOFONTINFO	Pointer to a font information structure
HVIO	Video handle

VioSetMode Returns an error/noerror code (USHORT).

This function is used to set the current display mode.

Parameters	Description
PVIOMODEINFO	Pointer to a mode information structure
HVIO	Video handle

VioSetOrg Returns an error/noerror code (USHORT).

This function is used to set the origin of the presentation space.

Parameters	Description
SHORT	The new origin row
SHORT	The new origin column
HVPS	Handle to a VIO presentation space

VioSetState Returns an error/noerror code (USHORT).
This function is used to set the video state.

Parameters	Description
PVOID	Pointer to the video state structure
HVIO	Video handle

VioShowBuf Returns an error/noerror code (USHORT).
This function is used to display a logical video buffer.

Parameters	Description
USHORT	Address within the video buffer to be displayed
USHORT	Length of data to be displayed
HVIO	Video handle

VioShowPS Returns an error/noerror code (USHORT).
This function is used to display a VIO presentation space.

Parameters	Description
SHORT	Rectangle depth
SHORT	Rectangle width
SHORT	Top left-hand corner cell offset
HVPS	Handle to a VIO presentation space

VioWrtCellStr Returns an error/noerror code (USHORT).
This function is used to write a string of character attributes to the display.

Parameters	Description
PCH	Pointer to the string to be written
USHORT	The length of the string to be written
USHORT	The starting row for the string
USHORT	The starting column for the string
HVIO	Video handle

VioWrtCharStr Returns an error/noerror code (USHORT).
This function is used to write a string to the screen.

Parameters	Description
PCH	Pointer to the string to be written
USHORT	The length of the string to be written
USHORT	The starting row for the string
USHORT	The starting column for the string
HVIO	Video handle

VioWrtCharStrAtt Returns an error/noerror code (USHORT).
This function is used to write a string with a repeated attribute.

Parameters	Description
PCH	Pointer to the string to be written
USHORT	The length of the string to be written
USHORT	Starting row for the string
USHORT	Starting column for the string
PBYTE	Attribute to be repeated
HVIO	Video handle

VioWrtNAttr Returns an error/noerror code (USHORT).

This function is used to write a specified attribute repeatedly.

Parameters	Description
PBYTE	Attribute to be repeated
USHORT	The number of times to repeat
USHORT	The starting row for the attribute
USHORT	The starting column for the attribute
HVIO	Video handle

VioWrtNCell Returns an error/noerror code (USHORT).

This function is used to write a character attribute pair a specified number of times.

Parameters	Description
PBYTE	Charattr pair to be repeated
USHORT	The number of times to repeat
USHORT	The starting row for the pair
USHORT	The starting column for the pair
HVIO	Video handle

VioWrtNChar Returns an error/noerror code (USHORT).

This function is used to repeatedly write a character to the screen.

Parameters	Description
PCH	Character to be repeated
USHORT	The number of times to repeat
USHORT	The starting row for the pair
USHORT	The starting column for the pair
HVIO	Video handle

VioWrtTTY Returns an error/noerror code (USHORT).

This function is used to write a character string to the screen at the current cursor.

Parameters	Description
PCH	The string to be written
USHORT	The length of the string
HVIO	Video handle

WinAddAtom Returns an atom (ATOM).

This function is used to add an atom to the atom table.

Parameters	Description
HATOMTBL	Handle to an atom table returned by WinCreateAtomTable
PSZ	String that composes the atom

WinAddProgram Returns a program handle (HPROGRAM).

This function is used to add a program to the Start Programs list.

Parameters	Description
HAB	Handle to an anchor block
PPIBSTRUCT	Pointer to the program information structure
HPROGRAM	Program group handle

WinAddSwitchEntry Returns a handle to the switch list (HSWITCH).

This function is used to add a program to the list of currently running programs.

Parameter	Description
PSWCNTRL	Pointer to the switch list data structure

WinAlarm Returns TRUE or FALSE (BOOL).

This function is used to sound one of three tones.

Parameters	Description
HWND	Window handle
USHORT	The alarm tone to sound

WinAllocMem Returns a pointer to the memory allocated (NPBYTE).

This function is used to allocate a block of memory from the heap.

Parameters	Description
HHEAP	Heap handle returned by WinCreateHeap
USHORT	Number of bytes to be allocated

WinAvailMem Returns the size of the largest free block of memory (USHORT).

This function is used to return the largest block of free memory available on the heap.

Parameters	Description
HHEAP	Heap handle returned by WinCreateHeap
BOOL	The indicator of whether to compact the heap
USHORT	Reserved (NULL)

WinBeginEnumWindows Returns an enumeration handle (HENUM).

This function is used to enable a program to enumerate all windows.

Parameter	Description
HWND	Window handle whose children will be enumerated

WinBeginPaint Returns a presentation space handle (HPS).

This function is used to enable a window to begin painting itself.

Parameters	Description
HWND	Handle of the window to be painted
HPS	The presentation space handle
PRECTL	Pointer to the bounding rectangle

WinBroadcastMsg Returns TRUE or FALSE (BOOL).

This function is used to broadcast a message to a set of windows.

Parameters	Description
HWND	Window handle of the parent
USHORT	Message identifier
MPARAM	Message parameter one
MPARAM	Message parameter two
USHORT	Indicator of whether the message should be sent or posted

WinCalcFrameRect Returns a success indicator (BOOL).

This function is used to calculate a frame rectangle size for a client or a client size for a frame.

Parameters	Description
HWND	Window handle of the frame
PRECTL	Pointer to a RECT structure
BOOL	TRUE indicates frame; FALSE indicates client-supplied

WinCallMsgFilter Returns the hook's return value (BOOL).

This function is used to execute a call to a message filter hook procedure.

Parameters	Description
HAB	The anchor block handle
PQMSG	Message to go to the hook procedure
USHORT	Filter code for the hook procedure

WinCancelShutdown Returns a success indicator (BOOL).

This function is used to cause the system to ignore this application when the system shuts down.

Parameters	Description
HMQ	Message queue handle of the program
BOOL	Flag to control cancel requests

WinCatch Returns whether the environment was saved (SHORT).

This function is used to save the current environment.

Parameter	Description
PCATCHBUF	Pointer to the buffer where the environment is saved

WinChangeSwitchEntry Returns 0 for success or an error code (USHORT).

This function is used to change an entry in the list of running programs.

Parameters	Description
HSWITCH	Handle to the switch list
PSWCNTRL	Pointer to the switch control structure

WinCloseClipbrd Returns a success/failure indicator (BOOL).

This function is used to close the clipboard.

Parameter	Description
HAB	The anchor block handle

WinCompareStrings Returns a comparison result (USHORT).

This function is used to compare two strings.

Parameters	Description
HAB	The anchor block handle
USHORT	Code page of the strings
USHORT	Country code of the strings
PSZ	First string
PSZ	Second string
USHORT	Reserved (NULL)

WinCopyAccelTable Returns the size of the table copied (USHORT).

This function is used to copy a specified accelerator table.

Parameters	Description
HACCEL	Handle to the accelerator table to be copied
PACCELTABLE	Pointer to an accelerator table structure
USHORT	The maximum size that can be copied

WinCopyRect Returns success/failure indicator (BOOL).

This function is used to copy a specified rectangle.

Parameters	Description
HAB	The anchor block handle
PRECTL	Rectangle to be copied to
PRECTL	Rectangle to be copied

WinCpTranslateChar Returns a character after translation (UCHAR).

This function is used to take a character and translate it to an equivalent character using the code page specified.

Parameters	Description
HAB	Anchor Block handle
USHORT	Original code page
UCHAR	The character to be translated
USHORT	Code page to translate to

WinCpTranslateString Returns a success/failure indicator (BOOL).

This function is used to translate a string to an equivalent string in another code page.

Parameters	Description
HAB	The anchor block handle
USHORT	Original code page
PSZ	String to be translated
USHORT	Code page to translate to
USHORT	Maximum length the result string may be
PSZ	The string after translation

WinCreateAccelTable Returns a handle to an accelerator table (HACCEL).

This function is used to create an accelerator table for an application.

APPENDIX A

Parameters	Description
HAB	The anchor block handle
PACCELTABLE	Pointer to an accelerator table structure

WinCreateAtomTable Returns a handle to an atom table (HATOMTBL).

This function is used to create an atom table for the application.

Parameters	Description
USHORT	Initial size of the table
USHORT	Size of the hash table for atoms

WinCreateCursor Returns a success/failure indicator (BOOL).

This function is used to create a cursor for use within a window.

Parameters	Description
HWND	Window in which the cursor will be used
SHORT	X position of the cursor
SHORT	Y position of the cursor
SHORT	Width of the cursor
SHORT	Height of the cursor
USHORT	Flags to control appearance
PRECTL	Pointer to a RECT structure for the cursor

WinCreateDataStructure Returns pass or fail status (BOOL).

This function is used to create a data structure in a standard form and returns a handle that is used to reference that structure.

Parameters	Description
HAB	The anchor block handle
USHORT	The number of elements in the structure
PSHORT	Pointer to the data types in the structure
USHORT	The structure length in bytes
PBYTE	Pointer to the buffer that contains the structure in application form
USHORT	The buffer length
HSTRUCT	The structure handle

WinCreateDlg Returns the new dialog's window handle (HWND).

This function is used to create a dialog window from a template structure.

Parameters	Description
HWND	Handle of the parent window
HWND	Handle of the owner window
PFNWP	Pointer to the dialog procedure
PDLGTEMPLATE	Pointer to the dialog template structure
PVOID	Pointer to the creation parameters

WinCreateFrameControls Returns a success/failure indicator (BOOL).

This function is used to create a set of frame controls for a window.

Parameters	Description
HWND	Handle of the window that gets the controls
PFRAMECDATA	Pointer to the structure holding the FCF flags
PSZ	String for the title bar

WinCreateGroup Returns a handle to the new group (HPROGRAM).

This function is used to create a new group in the program starter; if the group already exists, a handle to the existing group is simply returned.

Parameters	Description
HAB	The anchor block handle
PSZ	The new group name
UCHAR	The control for whether the group is visible
ULONG	Reserved field (NULL)
ULONG	Reserved field (NULL)

WinCreateHeap Returns a handle to the new heap (HHEAP).

This function is used to create a memory area for use by the application.

Parameters	Description
USHORT	Selector where the heap will be located
USHORT	The initial size of the heap
USHORT	Heap growth increments
USHORT	Size of the smallest heap element
USHORT	Size of the largest heap element
USHORT	Option flags

WinCreateMenu Returns a handle to the new menu window (HWND).

This function is used to create a new menu window.

Parameters	Description
HWND	Parent and owner window
PVOID	Pointer to the menu structure

WinCreateMsgQueue Returns a handle to a message queue (HMQ).

This function is used to create a new message queue for the current thread.

Parameters	Description
HAB	The anchor block handle
SHORT	The new queue's size (in messages)

WinCreatePointer Returns a handle to the new pointer (HPOINTER).

This function is used to take an icon and create a pointer for the application's use.

Parameters	Description
HWND	The desktop window handle
HBITMAP	Handle to the bitmap that will be the pointer
BOOL	Sizing flag
SHORT	X offset of the pointer hotspot
SHORT	X offset of the pointer hotspot

WinCreateStdWindow Returns a handle to the newly created frame window (HWND).
This function is used to create a standard window with a client area of the class specified.

Parameters	Description
HWND	Window handle of the parent
ULONG	Style flags for the frame window
PULONG	Flags for the frame controls
PSZ	Class name of the client
PSZ	Text string for the title bar
ULONG	Style flags for the client
HMODULE	Resource identifier
USHORT	Frame window identifier
PHWND	Window handle of the client returned

WinCreateWindow Returns a handle to the newly created window.
This function is used to create a simple window.

Parameters	Description
HWND	Window handle of the parent
PSZ	Class name of the window
PSZ	Text string for the window
ULONG	Style flags for the window
SHORT	X position (lower left)
SHORT	Y position (lower left)
SHORT	Width of window
SHORT	Height of window
HWND	Handle of the owner window
HWND	Handle of a sibling
USHORT	Window identifier
PVOID	Flags for control windows
PVOID	Presentation parameters (NULL)

WinDdeInitiate Returns a success/failure indicator (BOOL).

This function is used to send out a message during attempts to initiate a DDE dialog with another application.

Parameters	Description
HWND	The client's window handle
PSZ	Application name for the requested server
PSZ	Topic of interest

WinDdePostMsg Returns a success/failure indicator (BOOL).

This function is used to send a DDE message to another application.

Parameters	Description
HWND	Window handle of the addressee
HWND	Window handle of the sender
USHORT	Message identifier
PDDESTRUCT	Pointer to a structure containing information
BOOL	The indicator of whether to retry if this message fails

WinDdeRespond Returns the reply from the message (MRESULT).

This function is called by an application as a positive result to a DDE initiate message.

Parameters	Description
HWND	Window handle of the DDE client
HWND	Window handle of the DDE server
PSZ	Application name of the server
PSZ	Topic to be "discussed"

WinDefAVioProc Returns the result of the message sent to it (MRESULT).

This function is used to perform the default processing for AVIO windows.

Parameters	Description
HWND	Window handle of the addressee
USHORT	Message identifier
MPARAM	Message parameter one
MPARAM	Message parameter two

WinDefDlgProc Returns the result of the message sent to it (MRESULT).

This function is used to perform the default processing for dialog windows.

Parameters	Description
HWND	Window handle of the addressee
USHORT	Message identifier
MPARAM	Message parameter one
MPARAM	Message parameter two

WinDefWindowProc Returns the result of the message sent to it (MRESULT).

This function is used to perform the default processing for general windows.

Parameters	Description
HWND	Window handle of the addressee
USHORT	Message identifier
MPARAM	Message parameter one
MPARAM	Message parameter two

WinDeleteAtom Returns zero if successful; otherwise, returns a value equal to ATOM parameter (ATOM).

This function is used to delete an atom from an atom table.

Parameters	Description
HATOMTBL	Handle of an atom table
ATOM	Atom to be deleted

WinDeleteLibrary Returns pass or fail status (BOOL).

This function is used to make a previously loaded library unavailable.

Parameters	Description
HAB	The anchor block handle
HLIB	The library handle to be deleted

WinDeleteProcedure Returns pass or fail status (BOOL).

This function is used to delete a window or library procedure.

Parameters	Description
HAB	The anchor block handle
PFNWP	Pointer to the window procedure to be deleted

WinDestroyAccelTable Returns pass or fail status (BOOL).

This function is used to destroy an accelerator table.

Parameter	Description
HACCEL	Handle of the accelerator table

WinDestroyAtomTable Returns zero if successful; otherwise, returns a value equal to the HATOMTBL parameter (HATOMTBL).

This function is used to destroy an atom table.

Parameter	Description
HATOMTBL	Handle of the atom table

WinDestroyCursor Returns pass or fail status (BOOL).

This function is used to destroy the cursor of a specified window.

Parameter	Description
HWND	Handle of the window that owns the cursor

WinDestroyHeap Returns NULL if successful; otherwise, returns the heap handle (HHEAP).

This function is used to destroy a heap.

Parameter	Description
HHEAP	Handle of the heap to be destroyed

WinDestroyMsgQueue Returns pass or fail status (BOOL).

This function is used to destroy the message queue.

Parameter	Description
HMQ	Handle of the message queue

WinDestroyPointer Returns pass or fail status (BOOL).

This function is used to destroy a pointer or an icon.

Parameter	Description
HPOINTER	Handle of the pointer or icon to be destroyed

WinDestroyWindow Returns pass or fail status (BOOL).

This function is used to destroy a window and its child windows.

Parameter	Description
HWND	Handle of the window to be destroyed

WinDismissDlg Returns pass or fail status (BOOL).

This function is used to hide a dialog window and return a call to WinProcessDlg or WinDlgBox.

Parameters	Description
HWND	Handle of the dialog window
USHORT	Return value for WinProcessDlg or WinDlgBox

WinDispatchMsg Returns the result of the message dispatched (MRESULT).

This function is used to invoke a window procedure.

Parameters	Description
HAB	The anchor block handle
PQMSG	Pointer to the message structure

WinDlgBox Returns the value from WinDismissDlg or an error (USHORT).

This function is used to load and process a modal dialog window.

Parameters	Description
HWND	Handle of the parent window of the modal dialog
HWND	Handle of the owner window of the modal dialog
PFNWP	Pointer to the dialog procedure
HMODULE	Pointer to the resource containing the dialog template
USHORT	ID of the dialog template
PVOID	Pointer to the dialog procedure data

WinDrawBitmap Returns pass or fail status (BOOL).

This function is used to draw a bitmap.

Parameters	Description
HPS	The presentation space handle in which the bitmap is drawn
HBITMAP	Handle of the bitmap to be drawn
PRECTL	Pointer to the structure defining the portion of the bitmap to be drawn
PPOINTL	Pointer to the structure containing the target for the bitmap in the presentation space
LONG	Foreground color
LONG	Background color
USHORT	Flags identifying how the bitmap is to be drawn

WinDrawBorder Returns pass or fail status (BOOL).

This function is used to draw the border and border interior of a rectangle.

Parameters	Description
HPS	The presentation space handle
PRECTL	Pointer to the structure containing the bounding rectangle for the border
SHORT	Width of vertical sides of the border rectangle
SHORT	Width of the horizontal sides of the border rectangle
LONG	Color of the edge of the border
LONG	Color of the interior of the border
USHORT	Flags controlling the way the border is drawn

WinDrawPointer Returns pass or fail status (BOOL).

This function is used to draw a pointer.

Parameters	Description
HPS	The presentation space handle
SHORT	X coordinate at which to draw the pointer
SHORT	Y coordinate at which to draw the pointer
HPOINTER	Handle to the pointer to be drawn
USHORT	Indicator to control shading

WinDrawText Returns the number of characters drawn (SHORT).

This function is used to draw a single line of text in a rectangle.

Parameters	Description
HPS	The presentation space handle
SHORT	Number of characters in the text string
PCH	Pointer to the text string
PRECTL	Pointer to the structure defining the rectangle
LONG	Foreground color
LONG	Background color
USHORT	Flags controlling how the text is drawn

WinEmptyClipbrd Returns pass or fail status (BOOL).

This function is used to remove and to free all handles to data in the clipboard.

Parameter	Description
HAB	The anchor block handle

WinEnablePhysInput Returns pass or fail status (BOOL).

This function is used to enable or disable queuing of physical input.

Parameters	Description
HWND	Handle of the desktop window
BOOL	Enabling indicator

WinEnableWindow Returns pass or fail status (BOOL).

This function is used to enable or disable a window.

Parameters	Description
HWND	Handle of the window to be set
BOOL	Enabling indicator

WinEnableWindowUpdate Returns pass or fail status (BOOL).

This function is used to set the window visibility state for subsequent drawing without causing any change to the window on the display device.

Parameters	Description
HWND	Handle of the window affected
BOOL	Visibility indicator

WinEndEnumWindows Returns pass or fail status (BOOL).

This function is used to end the enumeration process for a given enumeration.

Parameter	Description
HENUM	Handle of the enumeration to be ended

WinEndPaint Returns pass or fail status (BOOL).

This function is used to indicate when the redrawing of a window is complete.

Parameter	Description
HPS	The presentation space handle

WinEnumClipbrdFmts Returns the next clipboard data format index (USHORT).

This function is used to enumerate the list of clipboard data formats available in the clipboard.

Parameters	Description
HAB	The anchor block handle
USHORT	Index of the previous clipboard data format

WinEnumDlgItem Returns the window handle of a dialog item (HWND).

This function is used to obtain the window handle of a dialog item.

Parameters	Description
HWND	Handle of the dialog window
HWND	Handle of the child window in the dialog
USHORT	Code for the item type
BOOL	Indicator specifying whether to lock the item window before the function is used to return the window handle

WinEqualRect Returns value indicating equality or inequality of two rectangles (BOOL).

This function is used to compare two rectangles.

Parameters	Description
HAB	The anchor block handle
PRECTL	Pointer to a structure describing rectangle one
PRECTL	Pointer to a structure describing rectangle two

WinExcludeUpdateRegion Returns a complexity value indicating the resulting form of the clipping area.

This function is used to subtract the update region of a window from the clipping region of a presentation space.

Parameters	Description
HPS	The presentation space handle
HWND	Handle of the window

WinFillRect Returns pass or fail status (BOOL).

This function is used to draw a filled rectangular area.

Parameters	Description
HPS	The presentation space handle
PRECTL	Pointer to the structure defining the rectangle to be filled in
LONG	Fill color

WinFindAtom Returns the value for the atom located (ATOM).

This function is used to find an atom in an atom table.

Parameters	Description
HATOMTBL	Handle of the atom table
PSZ	Pointer to the name of the atom to be found

WinFlashWindow Returns pass or fail status (BOOL).

This function is used to start or stop a window flashing.

Parameters	Description
HWND	Handle of the window
BOOL	Indicator of the desired action

WinFocusChange Returns pass or fail status (BOOL).

This function is used to change the focus window.

Parameters	Description
HWND	Handle of the desktop window
HWND	Handle of window to receive the focus
USHORT	Focus change indicators

WinFreeErrorInfo Returns pass or fail status (BOOL).

This function is used to free the memory allocated for an error information block.

Parameter	Description
PERRINFO	Pointer to the error information block that is to be freed

WinFreeMem Returns NULL if successful; otherwise, returns a value equal to the NPBYTE parameter (NPBYTE).

This function is used to free heap memory.

Parameters	Description
HHEAP	Handle of the heap
NPBYTE	Memory block to be freed
USHORT	Size of the memory to free

WinFreeMsg Returns pass or fail status (BOOL).

This function is used to destroy the data structures associated with the specified message.

Parameters	Description
HAB	The anchor block handle
HWND	The window handle associated with the message
USHORT	The message identity
MPARAM	The message parameter one
MPARAM	The message parameter two
MRESULT	The message reply

WinGetClipPS Returns a handle to a presentation space (HPS).

This function is used to get a clipped presentation space.

Parameters	Description
HWND	Window handle for which presentation space is needed
HWND	Window handle for clipping
USHORT	Clip flags

WinGetCurrentTime Returns the system timer count (ULONG).

This function is used to get the current time.

Parameter	Description
HAB	The anchor block handle

WinGetDlgMsg Returns pass or fail status (BOOL).

This function is used to obtain a message from the application's queue associated with the specified dialog.

Parameters	Description
HWND	The dialog window handle
PQMSG	Pointer to the message structure

WinGetErrorInfo Returns error information pointer or null (PERRINFO).

This function is used to obtain the pointer to an error information block.

Parameter	Description
HAB	The anchor block handle

WinGetKeyState Returns the state of the key (SHORT).

This function is used to obtain the state of a key at the time that the last message obtained from the queue was posted.

Parameters	Description
HWND	Handle of the desktop window
SHORT	Virtual key whose state is requested

WinGetLastError Returns the last nonzero error code (ERRORID).

This function is used to obtain the last nonzero error code set by a Presentation Manager function.

Parameter	Description
HAB	The anchor block handle

WinGetMinPosition Returns pass or fail status (BOOL).

This function is used to obtain the position to which a window is minimized.

Parameters	Description
HWND	Handle of the frame window
PSWP	Pointer to a structure in which the minimized window position will be stored
PPOINTL	Pointer to a value representing the point to which the window is to be minimized

WinGetMsg Returns an indicator of a WM_QUIT message (BOOL).

This function is used to obtain a message from the thread's message queue.

Parameters	Description
HAB	The anchor block handle
PQMSG	Pointer to a structure in which the message will be stored
HWND	Handle of the window for which messages are restricted

USHORT	First message identity
USHORT)	Last message identity

WinGetNextWindow Returns next window handle in enumeration list (HWND).
This function is used to obtain the handle of next window in an enumeration list.

Parameter	Description
HENUM	Enumeration handle

WinGetPhysKeyState Returns the key state (SHORT).
This function is used to obtain the physical state of a specified key.

Parameters	Description
HWND	Handle of the desktop window
SHORT	Hardware scan code of the key requested

WinGetPS Returns handle of a presentation space (HPS).
This function is used to obtain a cache presentation space.

Parameter	Description
HWND	Handle of window for which the presentation space is requested

WinGetScreenPS Returns a presentation space handle (HPS).
This function is used to obtain a presentation space.

Parameter	Description
HWND	Handle of the desktop window

WinGetSysBitmap Returns a system bitmap handle (HBITMAP).
This function is used to obtain a handle to a system bitmap.

Parameters	Description
HWND	Handle of the desktop window
USHORT	Index value of a system bitmap

WinInflateRect Returns pass or fail status (BOOL).
This function is used to expand a rectangle.

Parameters	Description
HAB	The anchor block handle
PRECTL	Pointer to a structure defining the rectangle to be expanded
SHORT	Horizontal expansion
SHORT	Vertical expansion

WinInitialize Returns a handle of an anchor block (HAB).
This function is used to initialize Presentation Manager facilities.

Parameter	Description
USHORT	Initialization options (NULL)

WinInSendMessage Returns a message-processing indicator (BOOL).

This function is used to determine whether the current thread is processing a message sent by another thread.

Parameter	Description
HAB	The anchor block handle

WinInstStartApp Returns application handle or null (HAPP).

This function is used to start a program from the list of installed programs.

Parameters	Description
HINI	The initialization file handle
HWND	The handle of the window to receive notification when the program terminates
USHORT	The number of elements in program parameter, must be one or two
PSZ far *	The array of program list information for program to be started
PSZ	The input parameters to be passed to the program
PVOID	Pointer to a reserved parameter that must be zero
USHORT	The options used to start the program

WinIntersectRect Returns intersection indicator (BOOL).

This function is used to calculate the intersection of two rectangles.

Parameters	Description
HAB	The anchor block handle
PRECTL	Pointer to a structure in which the values for rectangle of intersection will be stored
PRECTL	Pointer to the structure defining the first rectangle
PRECTL	Pointer to the structure defining the second rectangle

WinInvalidateRect Returns pass or fail status (BOOL).

This function is used to add a rectangle to a window's update region.

Parameters	Description
HWND	Handle of the window whose update region is to be modified
PRECTL	Pointer to a structure containing the definition of the rectangle to be added
BOOL	Indicator for descendants of window

WinInvalidateRegion Returns pass or fail status (BOOL).

This function is used to add a region to a window's update region.

Parameters	Description
HWND	Handle of the window whose update region is to be modified
HRGN	Handle of the region to be added to the window's update region
BOOL	Indicator for descendants of window

WinInvertRect Returns pass or fail status (BOOL).

This function is used to invert a rectangular area.

Parameters	Description
HPS	The presentation space handle
PRECTL	Pointer to the structure containing the definition of the rectangle to be inverted

WinIsChild Returns an indicator of a parent-child relationship (BOOL).

This function is used to determine whether a window is a descendant of another window.

Parameters	Description
HWND	Handle of the suspected child window
HWND	Handle of the parent window

WinIsRectEmpty Returns an indicator of empty or an error (BOOL).

This function is used to determine whether a rectangle is empty.

Parameters	Description
HAB	The anchor block handle
PRECTL	Pointer to a structure defining the rectangle to be examined

WinIsThreadActive Returns an active window indicator (BOOL).

This function is used to determine whether the active window belongs to the calling thread.

Parameter	Description
HAB	The anchor block handle

WinIsWindow Returns a validity indicator (BOOL).

This function is used to determine whether a window handle is valid.

Parameters	Description
HAB	The anchor block handle
HWND	Window handle to be examined

WinIsWindowEnabled Returns enabled state indicator (BOOL).

This function is used to determine the enabled state of a window.

Parameter	Description
HWND	Handle of the window

WinIsWindowShowing Returns pass or fail status (BOOL).

This function is used to determine whether a part of a window is visible to the user.

Parameter	Description
HWND	The window handle

WinIsWindowVisible Returns visibility state indicator (BOOL).

This function is used to determine the visibility state of a window.

Parameter	Description
HWND	Handle of the window

WinLoadAccelTable Returns an accelerator table handle (HACCEL).

This function is used to load an accelerator table.

Parameters	Description
HAB	The anchor block handle
HMODULE	Handle of the module containing the resource
USHORT	ID of the accelerator table

WinLoadDlg Returns the dialog window handle (HWND).

This function is used to create a dialog window.

Parameters	Description
HWND	Handle of the parent window of the dialog window
HWND	Handle of the owner window of the dialog window
PFNWP	Pointer to the dialog procedure for the dialog window
HMODULE	Handle of the module containing the dialog template
USHORT	ID of the dialog template
PVOID	Reserved (NULL)

WinLoadLibrary Returns a library handle or null (HLIB).

This function is used to make the library available to the application.

Parameters	Description
HAB	The anchor block handle
PSZ	Pointer to library name string

WinLoadMenu Returns the handle of the menu window (HWND).

This function is used to create a menu window from a menu template.

Parameters	Description
HWND	Handle of the parent and owner window
HMODULE	Handle of module containing the menu template
USHORT	ID of the menu template

WinLoadPointer Returns pointer handle (HPOINTER).

This function is used to load a pointer from a resource file.

Parameters	Description
HWND	Handle of the desktop window
HMODULE	Handle of the module containing the resource
USHORT	ID of the pointer to be loaded

WinLoadProcedure Returns the window procedure address or null (PFNWP).

This function is used to load a window or dialog procedure.

Parameters	Description
HAB	The anchor block handle
HLIB	The dynamic link library handle the procedure is in
PSZ	The name of the procedure

WinLoadString Returns length of the string loaded (SHORT).

This function is used to load a string from a resource.

Parameters	Description
HAB	The anchor block handle
HMODULE	Handle of the module containing the resource
USHORT	ID of the string to be loaded
SHORT	Size of buffer that will receive the string
PSZ	Pointer to buffer that will receive the string

WinLockHeap Returns the segment containing the passed heap (PVOID).
This function is used to lock a heap.

Parameter	Description
HHEAP	Handle to the heap to be locked

WinLockVisRegions Returns pass or fail status (BOOL).
This function is used to lock or unlock the visible regions of all windows on the screen.

Parameters	Description
HWND	Handle of the desktop window
BOOL	Indicator of action

WinLockWindow Returns the locked window handle (HWND).
This function is used to lock or unlock a window.

Parameters	Description
HWND	Handle of the window
BOOL	Indicator of action

WinLockWindowUpdate Returns a success indicator (BOOL).
This function is used to enable or disable output to a window.

Parameters	Description
HWND	Handle of the desktop window
HWND	Handle of the window to be acted upon

WinMakePoints Returns pass or fail status (BOOL).
This function is used to convert points to graphics points.

Parameters	Description
HAB	The anchor block handle
PWPOINT	Pointer to a structure containing the points to be converted
USHORT	Number of points to be converted

WinMakeRect Returns pass or fail status (BOOL).
This function is used to convert a rectangle to a graphics rectangle.

Parameters	Description
HAB	The anchor block handle
PWRECT	Pointer to a structure containing the definition of the rectangle to be converted

WinMapDlgPoints Returns pass or fail status (BOOL).
This function is used to map points between dialog coordinates and window coordinates.

Parameters	Description
HWND	Handle of the dialog window
PPOINTL	Pointer to the structure containing the points to be mapped
USHORT	Number of coordinate points
BOOL	Indicator for direction of calculation

WinMapWindowPoints Returns pass or fail status (BOOL).

This function is used to map a set of points from a coordinate space relative to one window into a coordinate space relative to another window.

Parameters	Description
HWND	Handle of the source window
HWND	Handle of the target window
PPOINTL	Pointer to the structure containing the points to be mapped
SHORT	Number of points to be mapped

WinMessageBox Returns a value indicating the user's response (USHORT).

This function is used to create and process a message box window.

Parameters	Description
HWND	Handle of the parent window of the message box
HWND	Handle of the owner window of the message box
PSZ	Pointer to the text to be placed in the message box window
PSZ	Pointer to the text for the title of the message box window
USHORT	ID of the message box window
USHORT	Window style of the message box window

WinModifyDataStructure Returns pass or fail status (BOOL).

This function is used to modify a structure stored in standard form.

Parameters	Description
HAB	The anchor block handle
USHORT	The number of elements in the structure
PSHORT	Pointer to the array of data types in the structure
USHORT	The buffer length in bytes
PBYTE	The buffer holding the structure in application form
HSTRUCT	The handle of the data structure to be modified

WinMsgMuxSemWait Returns an error code (USHORT).

This function is used to wait for any of a list of semaphores or for a message.

Parameters	Description
PUSHORT	Pointer to where the index of the cleared semaphore will be stored
PVOID	Pointer to list of semaphores
LONG	Function timeout value

WinMsgSemWait Returns an error code (USHORT).

This function is used to wait for a single semaphore or a message.

Parameters	Description
HWND	Handle of the dialog window
PPOINTL	Pointer to the structure containing the points to be mapped
USHORT	Number of coordinate points
BOOL	Indicator for direction of calculation

WinMapWindowPoints Returns pass or fail status (BOOL).

This function is used to map a set of points from a coordinate space relative to one window into a coordinate space relative to another window.

Parameters	Description
HWND	Handle of the source window
HWND	Handle of the target window
PPOINTL	Pointer to the structure containing the points to be mapped
SHORT	Number of points to be mapped

WinMessageBox Returns a value indicating the user's response (USHORT).

This function is used to create and process a message box window.

Parameters	Description
HWND	Handle of the parent window of the message box
HWND	Handle of the owner window of the message box
PSZ	Pointer to the text to be placed in the message box window
PSZ	Pointer to the text for the title of the message box window
USHORT	ID of the message box window
USHORT	Window style of the message box window

WinModifyDataStructure Returns pass or fail status (BOOL).

This function is used to modify a structure stored in standard form.

Parameters	Description
HAB	The anchor block handle
USHORT	The number of elements in the structure
PSHORT	Pointer to the array of data types in the structure
USHORT	The buffer length in bytes
PBYTE	The buffer holding the structure in application form
HSTRUCT	The handle of the data structure to be modified

WinMsgMuxSemWait Returns an error code (USHORT).

This function is used to wait for any of a list of semaphores or for a message.

Parameters	Description
PUSHORT	Pointer to where the index of the cleared semaphore will be stored
PVOID	Pointer to list of semaphores
LONG	Function timeout value

WinMsgSemWait Returns an error code (USHORT).

This function is used to wait for a single semaphore or a message.

Parameters	Description
HSEM	Handle of the semaphore
LONG	Function timeout value

WinMultWindowFromIDs Returns number of window handles returned (SHORT).

This function is used to find handles of child windows given a range of IDs.

Parameters	Description
HWND	Handle of the parent window
PHWND	Pointer to a structure in which the requested child window handles will be stored
USHORT	ID of the first child window in the range
USHORT	ID of the last child window in the range

WinNextChar Returns the next character in a string (PSZ).

This function is used to obtain the next character in a string.

Parameters	Description
HAB	The anchor block handle
USHORT	Code page
USHORT	Country code
PSZ	Pointer to the current character in the string

WinOffsetRect Returns pass or fail status (BOOL).

This function is used to offset a rectangle.

Parameters	Description
HAB	The anchor block handle
PRECTL	Pointer to a structure defining the rectangle to be offset
SHORT	X value of offset
SHORT	Y value of offset

WinOpenClipbrd Returns pass or fail status (BOOL).

This function is used to open the clipboard.

Parameter	Description
HAB	The anchor block handle

WinOpenWindowDC Returns a handle to a device context (HDC).

This function is used to open a device context for a window.

Parameter	Description
HWND	Handle of the window

WinPeekMsg Returns an indicator of whether a message is available (BOOL).

This function is used to get a message from the message queue, with the option of leaving the message in the queue.

Parameters	Description
HAB	The anchor block handle
PQMSG	Pointer to a structure in which the message will be stored
HWND	Handle of the window for which messages are restricted

USHORT	First message ID
USHORT	Last message ID
USHORT	Options to remove message from queue

WinPostMsg Returns pass or fail indicator (BOOL).

This function is used to post a message to a message queue associated with a window.

Parameters	Description
HWND	Handle of the window being addressed
USHORT	ID of the message
MPARAM	First message parameter
MPARAM	Second message parameter

WinPostQueueMsg Returns pass or fail status (BOOL).

This function is used to post a message to a message queue.

Parameters	Description
HMQ	Handle of the message queue
USHORT	ID of the message
MPARAM	First message parameter
MPARAM	Second message parameter

WinPrevChar Returns the previous character in a string (PSZ).

This function is used to obtain the previous character in a string.

Parameters	Description
HAB	The anchor block handle
USHORT	Code page
USHORT	Country code
PSZ	Pointer to the string
PSZ	Pointer to the current character in the string

WinProcessDlg Returns the value set by WinDismissDlg (USHORT).

This function is used to process messages for a dialog window in a modal manner.

Parameter	Description
HWND	Handle of the dialog window

WinPtInRect Returns a success indicator (BOOL).

This function is used to determine whether a point lies within a rectangle.

Parameters	Description
HAB	The anchor block handle
PRECTL	Pointer to the structure defining the rectangle
PPOINTL	Pointer to the structure containing the point to be determined

WinQueryAccelTable Returns a handle to an accelerator table (HACCEL).

This function is used to get a handle to an accelerator table for use by the program.

Parameters	Description
HAB	The anchor block handle
HWND	Handle of the frame window

WinQueryActiveWindow Returns a window handle (HWND).

This function is used to determine which window is currently active.

Parameters	Description
HWND	Handle of the parent window of the active window
BOOL	Window lock state flag

WinQueryAtomLength Returns the length of the atom (USHORT).

This function is used to determine the length of a particular atom's string.

Parameters	Description
HATOMTBL	Handle to an atom table
ATOM	Atom whose length is being queried

WinQueryAtomName Returns the length of the atom name (USHORT).

This function is used to return the name of a specified atom.

Parameters	Description
HATOMTBL	Handle to an atom table
ATOM	The atom whose name is being queried
PSZ	String buffer in which the name will be placed
USHORT	Length of the buffer (in bytes)

WinQueryAtomUsage Returns the usage count of the atom (USHORT).

This function is used to tell how many times a particular atom is used.

Parameters	Description
HATOMTBL	Handle to an atom table
ATOM	Atom whose use count is being queried

WinQueryBits Returns pass or fail status (BOOL).

This function will query the bits in a 4-byte integer and return the bits as an array of 32 integers.

Parameters	Description
HAB	The anchor block handle
LONG	The integer whose bits are to be queried
PLONG	Pointer to the 32-integer array

WinQueryBitsUnderMask Returns pass or fail status (BOOL).

This function queries bits in a four-byte integer under the control of a mask. The bits identified by the mask are returned as the bit value.

Parameters	Description
HAB	The anchor block handle
LONG	The integer whose bits are to be queried
LONG	The mask identifying the bits to be queried
PLONG	The bit value returned

WinQueryCapture Returns a window handle (HWND).

This function is used to determine which window (if any) has the mouse capture.

Parameters	Description
HWND	Window handle of the desktop
BOOL	Flag whether or not the window is locked

WinQueryClassInfo Returns a class existence flag (BOOL).
This function is used to return information about a specified class.

Parameters	Description
HAB	The anchor block handle
PSZ	Name of the class to be queried
PCLASSINFO	Pointer to a class information structure

WinQueryClassName Returns the class name's length (SHORT).
This function is used to return the class name of a specified window.

Parameters	Description
HWND	Window handle whose class is desired
SHORT	Length of the buffer in which the class name will be placed
PCH	Buffer for the class name

WinQueryClipbrdData Returns a handle to the clipboard data (ULONG).
This function is used to get a handle to the data with a certain data format in the clipboard.

Parameters	Description
HAB	The anchor block handle
USHORT	Clipboard data format desired

WinQueryClipbrdFmtInfo Returns a flag about whether the format exists (BOOL).
This function tells whether a certain data format exists in the clipboard and returns information about the format if it does exist.

Parameters	Description
HAB	The anchor block handle
USHORT	Data format desired
PUSHORT	Data format memory model and usage flags

WinQueryClipbrdOwner Returns a window handle (HWND).
This function is used to determine which window is the current owner of the clipboard.

Parameters	Description
HAB	The anchor block handle
BOOL	Window lock flag of the owner window

WinQueryClipbrdViewer Returns a window handle (HWND).
This function is used to tell which window is a current clipboard viewer.

Parameters	Description
HAB	The anchor block handle
BOOL	Window lock flag of the viewer window

WinQueryCp Returns the current code page (USHORT).
This function is used to determine the code page for a message queue.

APPENDIX A

Parameter	Description
HMQ	Handle to the message queue

WinQueryCpList Returns the number of available code pages (USHORT).

This function is used to return the list of available code pages.

Parameters	Description
HAB	The anchor block handle
USHORT	The maximum number of code pages
PUSHORT	Pointer to the list of code pages

WinQueryCursorInfo Returns a cursor existence flag (BOOL).

This function is used to provide information about a current cursor.

Parameters	Description
HWND	Handle to the desktop window
PCURSORINFO	Pointer to a cursor information structure

WinQueryDataStructure Returns pass or fail status (BOOL).

This function is used to obtain alias handles for components of a standard form structure.

Parameters	Description
HAB	The anchor block handle
HSTRUCT	The data structure handle
USHORT	The count of structure elements
PSHORT	The array of data types in the structure
USHORT	The buffer length
PBYTE	The buffer used to hold the structure in application form
USHORT	The length of the output structure

WinQueryDefinition Returns zero if error or length of details (USHORT).

This function is used to obtain the program details for a group or a program.

Parameters	Description
HAB	The anchor block handle
HPROGRAM	The program handle for requested information
PPIBSTRUCT	Pointer to structure buffer to receive data
USHORT	The buffer length in bytes

WinQueryDesktopWindow Returns a window handle (HWND).

This function is used to return a handle to the desktop window.

Parameters	Description
HAB	The anchor block handle
HDC	Handle to a device context

WinQueryDlgItemShort Returns a success/failure indicator (BOOL).

This function is used to get the text of a dialog item and convert it into an integer value.

Parameters	Description
HWND	Window handle of the parent
USHORT	ID of the window whose text is being converted

PSHORT Pointer to the resultant integer
BOOL Flag about whether the integer is signed

WinQueryDlgItemText Returns the length of the returned string (USHORT).
This function is used to determine the length of the text string in a dialog item.

Parameters	Description
HWND	Window handle of the parent
USHORT	ID of the window whose text is being converted
SHORT	Maximum length of the text
PSZ	Returned text string

WinQueryDlgItemTextLength Returns the length of a dialog item's text (SHORT).
This function is used to determine the length of a dialog window's text.

Parameters	Description
HWND	Window handle of the parent
USHORT	ID of the window whose text is being converted

WinQueryFocus Returns a window handle (HWND).
This function is used to determine which window has the focus.

Parameters	Description
HWND	Handle to the desktop window
BOOL	Flag for window locks

WinQueryMsgPos Returns a success/failure indicator (BOOL).
This function is used to determine the pointer position when the last message to the message queue has been posted.

Parameters	Description
HAB	The anchor block handle
PPOINTL	Pointer to the pointer position values

WinQueryMsgTime Returns the message time (ULONG).
This function is used to determine the message time for the last message received.

Parameter	Description
HAB	The anchor block handle

WinQueryObjectWindow Returns a window handle (HWND).
This function is used to provide the object window's handle.

Parameter	Description
HWND	Handle of the desktop window

WinQueryPointer Returns a pointer handle (HPOINTER).
This function is used to provide the desktop's pointer handle.

Parameter	Description
HWND	Handle of the desktop window

WinQueryPointerInfo Returns a success/failure indicator (BOOL).
This function is used to provide information about a specific pointer.

Parameters	Description
HPOINTER	Handle to the pointer
PPOINTERINFO	Pointer to a pointer information structure

WinQueryPointerPos Returns a success/failure indicator (BOOL).
This function is used to determine the position of the pointer.

Parameters	Description
HWND	Handle of the desktop window
PPOINTL	Pointer to the pointer position

WinQueryProfileData Returns a success/failure indicator (BOOL).
This function is used to get a string from the OS2 INI file.

Parameters	Description
HAB	The anchor block handle
PSZ	Program name string
PSZ	Key for which text is desired
PVOID	The returned data
PUSHORT	The size of the returned data

WinQueryProfileInt Returns the queried integer (SHORT).
This function is used to get an integer from a program's profile string.

Parameters	Description
HAB	The anchor block handle
PSZ	Program name string
PSZ	Key for which integer is desired
SHORT	Default value if program cannot be found

WinQueryProfileSize Returns a success/failure code (USHORT).
This function is used to get the size of a specified entry from an OS2 INI.

Parameters	Description
HAB	The anchor block handle
PSZ	Program name string
PSZ	Key for which profile length is desired
PUSHORT	Size of the returned data

WinQueryProfileString Returns the length of the string (USHORT).
This function is used to get a text string from OS2 INI.

Parameters	Description
HAB	The anchor block handle
PSZ	Program name string
PSZ	Key for which text is desired
PSZ	Default string if requested one is not found
PVOID	The returned data
USHORT	Maximum size of the returned data

WinQueryProgramTitles Returns pass or fail status (BOOL).

This function is used to obtain information about a program or a group defined in the program list.

Parameters	Description
HAB	The anchor block handle
HPROGRAM	The requested program or group handle
PPROGRAMENTRY	The buffer address to receive information
USHORT	The buffer length
PUSHORT	The address to receive the title count

WinQueryQueueInfo Returns a success/failure indicator (BOOL).

This function is used to provide information about a specified message queue.

Parameters	Description
HMQ	Handle to the message queue
PMQINFO	Pointer to a message queue information structure
USHORT	The length of the message queue information structure (in bytes)

WinQueryQueueStatus Returns a set of queue status bits (ULONG).

This function is used to tell information about the caller's message queue.

Parameter	Description
HWND	Handle of the desktop window

WinQuerySessionTitle Returns pass or fail status (USHORT).

This function is used to determine the title under which a program was started or added to the switch list.

Parameters	Description
HAB	The anchor block handle
USHORT	The session ID
PSZ	Pointer to the string containing the program title
USHORT	The title length

WinQuerySwitchEntry Returns pass or fail status (USHORT).

This function is used to obtain a copy to the task list data for a program.

Parameters	Description
HSWITCH	The switch list entry handle
PSWCNTRL	Pointer to the structure containing the switch list information

WinQuerySwitchHandle Returns a program's switch list handle or null (HSWITCH).

This function is used to obtain the switch list handle for a window.

Parameters	Description
HWND	The window handle of a program
PID	The process ID of the program

WinQuerySwitchList Returns the number of entries in the switch list or error status (USHORT).

This function is used to obtain information about entries in the switch list.

APPENDIX A

Parameters	Description
HAB	The anchor block handle
PSWBLOCK	Pointer to the switch block structure holding the description of the switch list entries
USHORT	The length of the switch block structure in bytes

WinQuerySysColor Returns the color value (LONG).

This function is used to inform the calling function of the current system color.

Parameters	Description
HWND	Handle of the desktop window
LONG	System color value (SYSCLR_*)
LONG	Reserved (NULL)

WinQuerySysModalWindow Returns a window handle (HWND).

This function is used to provide the handle of the current system modal window.

Parameters	Description
HWND	Handle of the desktop window
BOOL	Window lock flag

WinQuerySysPointer Returns a handle to a pointer (HPOINTER).

This function is used to give the handle to the current system pointer.

Parameters	Description
HWND	Handle of the desktop window
SHORT	ID of the system pointer
BOOL	Copy flag

WinQuerySystemAtomTable Returns a handle to an atom table (HATOMTBL).

This function is used to provide the handle of the system's atom table.

Parameter	Description
VOID	The function takes no parameters

WinQuerySysValue Returns a set of system value (SV_*) flags (LONG).

This function is used to determine what system value flags are set.

Parameters	Description
HWND	Handle of the desktop window
SHORT	System value ID

WinQueryTaskTitle Returns success/failure code (USHORT).

This function is used to get the string under which this program was started.

Parameters	Description
USHORT	Session ID
PSZ	Title under which the program was started
USHORT	Length of data return buffer (in bytes)

WinQueryUpdateRect Returns a success/failure flag (BOOL).

This function is used to return the rectangle that needs to be repainted.

Parameters	Description
HWND	Handle of window whose update region is desired
PRECTL	Update region

WinQueryUpdateRegion Returns region flags (SHORT).

This function is used to get a window's update region.

Parameters	Description
HWND	Handle of the window whose update region is desired
HRGN	Handle to the update region

WinQueryValue Returns pass or fail status (BOOL).

This function is used to decompose a four-byte integer into one- or two-byte integers or bit values.

Parameters	Description
HAB	The anchor block handle
LONG	The four-byte integer to be decomposed
LONG	The number of elements to be extracted
PLONG	Pointer to an array containing the data types of the components
PLONG	Pointer to the array of components extracted

WinQueryVersion Returns version information (ULONG).

This function is used to return version information flags.

Parameter	Description
HAB	The anchor block handle

WinQueryWindow Returns a window handle (HWND).

This function is used to get the window handle of the window that has a specific relationship to the window provided.

Parameters	Description
HWND	Window handle to be queried
SHORT	Relationship specification flags
BOOL	The indicator of whether the window is locked

WinQueryWindowDC Returns a device context handle (HDC).

This function is used to get a handle to the device context for a specified window.

Parameter	Description
HWND	Window handle whose device context is desired

WinQueryWindowLockCount Returns the window lock count (SHORT).

This function is used to return the lock count for a specified window.

Parameter	Description
HWND	Window handle whose lock count is desired

WinQueryWindowPos Returns a success/failure indicator (BOOL).

This function is used to return a window's position and size.

Parameters	Description
HWND	Window handle
PSWP	Pointer to an SWP structure to determine position and size

WinQueryWindowProcess Returns a success/failure indicator (BOOL).

This function is used to get the thread and process IDs under which the window is running.

Parameters	Description
HWND	Window handle to be queried
PPID	Pointer to the PID structure where the process ID will be returned
PTID	Pointer to the TID structure where the thread ID will be returned

WinQueryWindowPtr Returns a pointer to the pointer's value (PVOID).

This function is used to get a pointer value for a window.

Parameters	Description
HWND	Window handle to be queried
SHORT	Index for the pointer to be obtained

WinQueryWindowRect Returns a success/failure indicator (BOOL).

This function is used to return a rectangle structure for a specified window.

Parameters	Description
HWND	Window handle to be queried
PRECTL	Pointer to a rectangle structure

WinQueryWindowText Returns length of the returned string (SHORT).

This function is used to get the text of the specified window.

Parameters	Description
HWND	Window handle to be queried
SHORT	Size of the buffer to which the text will be sent
PCH	Buffer for the text

WinQueryWindowTextLength Returns the length of the text (SHORT).

This function is used to query the length of a window's text.

Parameter	Description
HWND	Window handle to be queried

WinQueryWindowULong Returns the window's ULONG value (ULONG).

This function is used to get a window's ULONG value from its window words.

Parameters	Description
HWND	Window handle to be queried
SHORT	Index flags

WinQueryWindowUShort Returns the window's USHORT value (USHORT).

This function is used to get a window's USHORT value from its window words.

Parameters	Description
HWND	Window handle to be queried
SHORT	Index flags

WinReallocMem Returns a pointer to the heap (NPBYTE).

This function is used to modify the allocation of memory on the heap.

Parameters	Description
HHEAP	Heap handle
NPBYTE	Block to be changed
USHORT	Previous size
USHORT	New size

WinRegisterClass Returns a success/failure indicator (BOOL).

This function is used to register a new window class.

Parameters	Description
HAB	The anchor block handle
PSZ	Newly registered class name
PFNWP	Pointer to the window procedure for the class
ULONG	Class style flags
USHORT	Size of window words (in bytes)

WinRegisterUserDatatype Returns pass or fail status (BOOL).

This function is used to register a data type and define its structure.

Parameters	Description
HAB	The anchor block handle
SHORT	The data type code to be defined
SHORT	The count of elements to be defined
PSHORT	Pointer to the structure components' data type codes

WinRegisterUserMsg Returns pass or fail status (BOOL).

This function is used to register a user message and define its parameters.

Parameters	Description
HAB	The anchor block handle
USHORT	The message ID
SHORT	The data type of message parameter one
SHORT	Indicates whether parameter one is for input, output, or both
SHORT	The data type of message parameter two
SHORT	Indicates whether parameter two is for input, output, or both
SHORT	The data type of the message reply

WinRegisterWindowDestroy Returns a success/failure indicator (BOOL).

This function is used to cause other windows to be notified when the specified window is destroyed.

Parameters	Description
HWND	Window handle
BOOL	Registration flags

WinReleaseHook Returns a success/failure indicator (BOOL).

This function is used to remove a previously installed hook.

Parameters	Description
HAB	The anchor block handle
HMQ	Handle of message queue from which hook is removed
SHORT	Type of hook
PFN	Pointer to the address of the hook procedure
HMODULE	Module handle (if the hook is in a DLL)

WinReleasePS Returns a success/failure indicator (BOOL).

This function is used to release a presentation space obtained with either WinGetPS or WinGetScreenPS.

Parameter	Description
HPS	Handle of the cached presentation space to be released

WinRemoveSwitchEntry Returns a success/failure indicator (USHORT).

This function is used to remove an entry from the task manager's switch list.

Parameter	Description
HSWITCH	Handle to the task manager's entry for the program

WinScrollWindow Returns region flags (SHORT).

This function is used to scroll a window by a specified amount.

Parameters	Description
HWND	Handle of the window to be scrolled
SHORT	Amount of horizontal scrolling (to the right)
SHORT	Amount of vertical scrolling (upward)
PRECTL	The actual scrolling rectangle
PRECTL	The clipping rectangle
HRGN	Handle to the update region
PRECTL	The update rectangle
USHORT	Scrolling flags

WinSendDlgItemMsg Returns the reply value from the message (MRESULT).

This function is used to send a message to the child of the parent specified with the ID of parameter two.

Parameters	Description
HWND	Handle of the parent window
USHORT	ID of the child
USHORT	Message ID
MPARAM	Message parameter one
MPARAM	Message parameter two

WinSendMsg Returns the reply value from the message (MRESULT).

This function is used to send a message to a specified window.

Parameters	Description
HWND	Window handle of the addressee
USHORT	Message identifier
MPARAM	Message parameter one
MPARAM	Message parameter two

WinSetAccelTable Returns a success/failure indicator (BOOL).

This function is used to set an accelerator table for a window or queue.

Parameters	Description
HAB	The anchor block handle
HACCEL	Handle to an accelerator table
HWND	Handle of the frame window

WinSetActiveWindow Returns a success/failure indicator (BOOL).

This function is used to set the active window.

Parameters	Description
HWND	Handle of the desktop window
HWND	Window handle whose main window is to be made active

WinSetBits Returns pass or fail status (BOOL).

This function is used to modify bits in a four-byte integer under the control of an array of 32 integers.

Parameters	Description
HAB	The anchor block handle
LONG	The integer whose bits are to be modified
PLONG	Pointer to the array of 32 integers used to modify the integer

WinSetBitsUnderMask Returns pass or fail status (BOOL).

This function is used to modify bits in a four-byte integer under the control of a mask; the bits identified by the mask are set equal to the corresponding bits in the bit value.

Parameters	Description
HAB	The anchor block handle
LONG	The integer whose bits are to be modified
LONG	The mask used to identify the bits to be modified
LONG	The bit value to be used to modify the integer

WinSetCapture Returns a success/failure indicator (BOOL).

This function is used to set the mouse capture.

Parameters	Description
HWND	Handle of the desktop window
HWND	Handle of the window that is getting the mouse capture

WinSetClassMsgInterest Returns a success/failure indicator (BOOL).

This function is used to set a window class to be interested/uninterested in only certain messages.

APPENDIX A

Parameters	Description
HAB	The anchor block handle
PSZ	Class name to be set
USHORT	Message class
SHORT	Interest flag

WinSetClipbrdData Returns a success/failure indicator (BOOL).

This function is used to place data in the clipboard area.

Parameters	Description
HAB	The anchor block handle
ULONG	Handle to the data being placed in the clipboard
USHORT	Data format flag
USHORT	Format information if it is user-defined

WinSetClipbrdOwner Returns a success/failure indicator (BOOL).

This function is used to set a new clipboard owner window.

Parameters	Description
HAB	The anchor block handle
HWND	Window handle of the new owner

WinSetClipbrdViewer Returns a success/failure indicator (BOOL).

This function is used to set a window to be the current clipboard viewer.

Parameters	Description
HAB	The anchor block handle
HWND	Window handle of the new viewer

WinSetCp Returns a success/failure indicator (BOOL).

This function is used to set the code page for a specified message queue.

Parameters	Description
HMQ	Handle to the message queue
USHORT	Code page to be set

WinSetDlgItemShort Returns a success/failure indicator (BOOL).

This function is used to convert an integer value into a text string for a dialog item.

Parameters	Description
HWND	Handle of the parent window
USHORT	ID of the child window
USHORT	Integer used for the item text
BOOL	Sign flag

WinSetDlgItemText Returns a success/failure indicator (BOOL).

This function is used to set a dialog window's text.

Parameters	Description
HWND	Handle of the parent window
USHORT	ID of the child window
PSZ	Text to set

WinSetFocus Returns a success/failure indicator (BOOL).

This function is used to set the window that will get the focus.

Parameters	Description
HWND	Handle of the desktop window
HWND	Window handle that will get the focus

WinSetHook Returns a success/failure indicator (BOOL).

This function is used to set a hook.

Parameters	Description
HAB	The anchor block handle
HMQ	Handle to the message queue that will get the hook
SHORT	Type of hook
PFN	Pointer to the hook procedure
HMODULE	Resource ID (if the procedure is in a DLL)

WinSetKeyboardStateTable Returns a success/failure indicator (BOOL).

This function is used to manipulate the keyboard state table for the current process.

Parameters	Description
HWND	Handle of the desktop window
PBYTE	Keyboard state table
BOOL	Set/unset flag

WinSetMsgInterest Returns a success/failure indicator (BOOL).

This function is used to set the message interest for a single window.

Parameters	Description
HWND	Window handle
USHORT	Message class to set interest for
SHORT	Interest flag

WinSetMsgMode Returns pass or fail status (BOOL).

This function is used to establish the mode for the generation and processing of messages for a program's window.

Parameters	Description
HAB	The anchor block handle
PSZ	Pointer to the window class name string
SHORT	The message mode ID

WinSetMultWindowPos Returns a success/failure indicator (BOOL).

This function is used to set many windows' positions at once.

Parameters	Description
HAB	The anchor block handle
PSWP	Pointer to an array of SWP structures for the windows to be set
USHORT	Number of SWP structures

WinSetOwner Returns a success/failure indicator (BOOL).

This function is used to set a new owner for a window.

Parameters	Description
HWND	Window handle whose owner is being changed
HWND	Handle of the new owner to be set

WinSetParent Returns a success/failure indicator (BOOL).

This function is used to set a new parent for a window.

Parameters	Description
HWND	Window handle whose parent is to be set
HWND	New parent window handle
BOOL	Redraw the window flag

WinSetPointer Returns a success/failure indicator (BOOL).

This function is used to set a new pointer for the desktop.

Parameters	Description
HWND	Handle of the desktop window
HPOINTER	Handle to the new pointer

WinSetPointerPos Returns a success/failure indicator (BOOL).

This function is used to change the pointer position.

Parameters	Description
HWND	Handle of the desktop window
SHORT	X coordinate of the pointer (in screen coordinates)
SHORT	Y coordinate of the pointer (in screen coordinates)

WinSetRect Returns a success/failure indicator (BOOL).

This function is used to set up a rectangle structure with the specified values.

Parameters	Description
HAB	The anchor block handle
PRECTL	Pointer to the rectangle structure to be set up
SHORT	Left value of rectangle
SHORT	Bottom value of rectangle
SHORT	Right value of rectangle
SHORT	Top value of rectangle

WinSetRectEmpty Returns a success/failure indicator (BOOL).

This function is used to blank out a rectangle structure.

Parameters	Description
HAB	The anchor block handle
PRECTL	Pointer to the rectangle structure to be blanked

WinSetSynchroMode Returns pass or fail status (BOOL).

This function is used to specify the current synchronization mode. This mode in turn determines how the distributed message queue is processed.

Parameters	Description
HAB	The anchor block handle
SHORT	The synchronization mode to be used

WinSetSysColors Returns a success/failure indicator (BOOL).
This function is used to set new system colors for the SYSCLR_* values.

Parameters	Description
HWND	Handle of the desktop window
ULONG	Coloring flags
ULONG	Format flags for the table
LONG	Beginning color index
ULONG	Number of elements in the color table
PLONG	Pointer to the color table

WinSetSysModalWindow Returns a success/failure indicator (BOOL).
This function is used to change the current system modal window.

Parameters	Description
HWND	Handle of the desktop window
HWND	Handle of the new system modal window

WinSetSysValue Returns a success/failure indicator (BOOL).
This function is used to set new system values (SV_*).

Parameters	Description
HWND	Handle of the desktop window
SHORT	System value ID
LONG	System value

WinSetValue Returns pass or fail status (BOOL).
This function is used to compose a four-byte integer composed of one-byte or two-byte integers or bit values.

Parameters	Description
HAB	The anchor block handle
LONG	The number of components in the integer
PLONG	Pointer to the component data types
PLONG	Pointer to the data components array
LONG	The four-byte integer

WinSetWindowBits Returns a success/failure indicator (BOOL).
This function is used to set bits into the window words of a specified window.

Parameters	Description
HWND	Window handle
SHORT	Index of where the bits are to be sent
ULONG	Data to be stored in the window words
ULONG	Bit mask to indicate which bits are to be written

APPENDIX A

WinSetWindowPos Returns a success/failure indicator (BOOL).

This function is used to change a window's position and size.

Parameters	Description
HWND	Window handle to move/size
HWND	The window the moved window will be moved behind
SHORT	New X coordinate
SHORT	New Y coordinate
SHORT	New width
SHORT	New height
USHORT	Positioning flags

WinSetWindowPtr Returns a success/failure indicator (BOOL).

This function is used to place a pointer into the specified window's window words.

Parameters	Description
HWND	Window handle
SHORT	Index into the window words
PVOID	Pointer to be stored

WinSetWindowText Returns a success/failure indicator (BOOL).

This function is used to change the text of a window.

Parameters	Description
HWND	Window handle whose text is to be changed
PSZ	New window text

WinSetWindowULong Returns a success/failure indicator (BOOL).

This function is used to place a ULONG in the window words of the specified window.

Parameters	Description
HWND	Window handle
SHORT	Index to the window words
ULONG	ULONG to go into the window words

WinSetWindowUShort Returns a success/failure indicator (BOOL).

This function is used to place a USHORT in the window words of the specified window.

Parameters	Description
HWND	Window handle
SHORT	Index into the window words
USHORT	USHORT to go into the window words

WinShowCursor Returns a success/failure indicator (BOOL).

This function is used to hide or display a cursor associated with the specified window.

Parameters	Description
HWND	Window handle that owns the cursor
BOOL	Show/hide flag

WinShowPointer Returns a success/failure indicator (BOOL).

This function is used to show or hide the pointer by updating the pointer display level.

Parameters	Description
HWND	Handle of the desktop window
BOOL	Level update flag (increment or decrement)

WinShowTrackRect Returns a success/failure indicator (BOOL).

This function is used to show or hide a tracking rectangle.

Parameters	Description
HWND	Window handle
BOOL	Show/hide flag

WinShowWindow Returns a success/failure indicator (BOOL).

This function is used to show or hide a window.

Parameters	Description
HWND	Window to be shown or hidden
BOOL	Show/hide flag

WinStartTimer Returns the timer ID (USHORT).

This function is used to start a timer to send WM_TIMER messages.

Parameters	Description
HAB	The anchor block handle
HWND	Window handle that is to get the WM_TIMER messages
USHORT	Timer ID
USHORT	Timer time (in milliseconds)

WinStopTimer Returns a success/failure indicator (BOOL).

This function is used to stop a previously created timer.

Parameters	Description
HAB	The anchor block handle
HWND	Window handle that has the timer
USHORT	Timer identifier

WinSubclassWindow Returns a pointer to the old window procedure (PFNWP).

This function is used to replace one window procedure with another for a specified window.

Parameters	Description
HWND	Handle of window that is having its window procedure replaced
PFNWP	Pointer to the new window procedure

WinSubstituteStrings Returns the number of characters returned (SHORT).

This function is used to replace specified text with other text.

Parameters	Description
HWND	Window handle processing the function
PSZ	Original string
SHORT	Maximum number of characters
PSZ	Final string

WinSubtractRect Returns a value for whether rectangle is empty (BOOL).
This function is used to subtract two rectangles.

Parameters	Description
HAB	The anchor block handle
PRECTL	Pointer to the resultant rectangle structure
PRECTL	Pointer to the first rectangle structure
PRECTL	Pointer to the second rectangle structure

WinSwitchToProgram Returns an error or zero for success (USHORT).
This function is used to make a specific program the active program.

Parameter	Description
HSWITCH	The program's switch list entry handle

WinTerminate Returns a success/failure indicator (BOOL).
This function is used to terminate registration with the Presentation Manager.

Parameter	Description
HAB	The anchor block handle

WinTerminateApp Returns pass or fail status (BOOL).
This function is used to terminate a child application.

Parameter	Description
HAPP	The application handle returned by WinInstStartApp

WinThrow Returns the return value for WinCatch (SHORT).
This function is used to restore an environment saved by WinCatch.

Parameter	Description
PCATCHBUF	Pointer to the environment save buffer

WinTrackRect Returns a success/failure indicator (BOOL).
This function is used to draw a tracking rectangle within the specified window.

Parameters	Description
HWND	Window handle inside which tracking occurs
HPS	The presentation space handle
PTRACKINFO	Pointer to a tracking information buffer

WinTranslateAccel Returns a success/failure indicator (BOOL).
This function is used to translate a WM_CHAR message into an accelerator key value.

Parameters	Description
HAB	The anchor block handle
HWND	Window handle where message is destined
HACCEL	Handle to an accelerator table
PQMSG	Pointer to the QMSG structure to be translated

WinUnionRect Returns whether the resultant rectangle is empty (BOOL).
This function is used to give a rectangle that bounds two specified rectangles.

Parameters	Description
HAB	The anchor block handle
PRECTL	Pointer to the resultant rectangle structure
PRECTL	Pointer to the first rectangle structure
PRECTL	Pointer to the second rectangle structure

WinUpdateWindow Returns a success/failure indicator (BOOL).

This function is used to update a window and its children.

Parameter	Description
HWND	Handle of the window to be updated

WinUpper Returns the length of the resultant string (USHORT).

This function is used to convert a string to uppercase, optionally using a different code page.

Parameters	Description
HAB	The anchor block handle
USHORT	Code page to be used (or NULL)
USHORT	Country code to be used (or NULL)
PSZ	String to be converted

WinUpperChar Returns the resultant character (USHORT).

This function is used to translate a specified character to uppercase, optionally using a different code page.

Parameters	Description
HAB	The anchor block handle
USHORT	Code page to be used (or NULL)
USHORT	Country code to be used (or NULL)
PSZ	Character to be converted

WinValidateRect Returns a success/failure indicator (BOOL).

This function is used to remove a specified rectangle from the update region of a window.

Parameters	Description
HWND	Window handle whose update region is changed
PRECTL	Pointer to a rectangle structure to be subtracted
BOOL	Include-descendants flag

WinValidateRegion Returns a success/failure indicator (BOOL).

This function is used to remove a specified region from the update region of a window.

Parameters	Description
HWND	Window handle whose update region is changed
PRECTL	Pointer to a region structure to be subtracted
BOOL	Include-descendants flag

WinWaitMsg Returns a success/failure indicator (BOOL).

This function is used to make the thread running with this HAB wait until a message within the specified range is received.

Parameters	Description
HAB	The anchor block handle
USHORT	ID of the first message in the range
USHORT	ID of the last message in the range

WinWindowFromDC Returns a window handle (HWND).

This function is used to provide a window handle associated with a specified device context.

Parameter	Description
HDC	Handle to a device context

WinWindowFromID Returns a window handle (HWND).

This function is used to provide a window handle given the parent of the target window and the window's ID.

Parameters	Description
HWND	Handle of the parent window
USHORT	ID of the desired child

WinWindowFromPoint Returns a window handle (HWND).

This function is used to provide a window handle for the window under a specified point on the screen that is a descendant of a specified window.

Parameters	Description
HWND	Window handle whose children are examined
PPOINTL	Pointer to a point structure to be tested
BOOL	Child enumeration flag
BOOL	Window lock flag

WinWriteProfileData Returns a success/failure indicator (BOOL).

This function is used to write a string for a program into the OS2 INI file.

Parameters	Description
HAB	The anchor block handle
PSZ	Application name
PSZ	Key name string
PVOID	Pointer to the key value
USHORT	Length of the key value data

WinWriteProfileString Returns a success/failure indicator (BOOL).

This function is used to write a key/value pair into OS2 INI.

Parameters	Description
HAB	The anchor block handle
PSZ	Application name
PSZ	Key name string
PSZ	Pointer to the key value

B

Structures

Each of the application program interfaces (APIs) uses a set of specific data formats to pass data to the DLLs that execute the function. In many cases, complex data types are used. The list of structures in this appendix contains their definitions implemented for the C language. The data type names should remain consistent across language implementations, but the underlying types change for each language. All of the types referenced in this book are contained in this list.

Organization of the Structures List

This list has been organized so you can read each structure and determine the prefix for variables declared with each type. In the case of data structures (as opposed to data types), the elements of each are listed under the structure. An example of each follows.

Example 1

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
BOOL	unsigned short	f

Example 1 says that the data type BOOL is an unsigned short value (as defined by the C language). The prefix that should be used for variables of the BOOL type is a lowercase *f*. This stands for a flag.

Therefore, a variable that begins with *f* is a boolean value (either TRUE or FALSE) and is an unsigned short value.

Example 2

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
ARCPARAMS	struct	arcp
IP	LONG	
IQ	LONG	
IR	LONG	
IS	LONG	

In this case, ARCPARAMS is a structure containing four elements. The elements are four LONG values. The names of the values are IP, IQ, IR, and IS, which represent the four parts of the arc parameters discussed in the chapter on graphics primitives.

As you can see, each of the four LONG values begins with an *I* to signify it is a LONG. The prefix used to identify an ARCPARAMS structure variable name is arcp.

So, if you see a variable referenced as arcpMyParams.IQ, you will know that is is the Q element of a set of arc parameters.

Alphabetical Listing of Structures

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
ACCEL	struct	acc
fs	USHORT	
key	USHORT	
cmd	USHORT	
ACCELTABLE	struct	acct
cAccel	USHORT	
codepage	USHORT	
aaccel[1]	ACCEL	
ARCPARAMS	struct	arcp
IP	LONG	
IQ	LONG	
IR	LONG	
IS	LONG	
AREABUNDLE	struct	pbnd
IColor	LONG	
IBackColor	LONG	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
usMixMode	USHORT	
usBackMixMode	USHORT	
usSet	USHORT	
usSymbol	USHORT	
ptlRefPoint	POINTL	
ATOM	USHORT	
BITMAPFILEHEADER	struct	bfh
usType	USHORT	
cbSize	ULONG	
xHotspot	INT	
yHotspot	INT	
offBits	ULONG	
bmp	BITMAPINFOHEADER	
BITMAPINFO	struct	bmi
cbFix	ULONG	
cx	USHORT	
cy	USHORT	
cPlanes	USHORT	
cBitCount	USHORT	
argbColor[1]	RGB	
BITMAPINFOHEADER	struct	bmp
cbFix	ULONG	
cx	USHORT	
cy	USHORT	
cPlanes	USHORT	
cBitCount	USHORT	
BOOL	unsigned short	f
BTNCDATA	struct	btncd
cb	USHORT	
fsCheckState	USHORT	
fsHiliteState	USHORT	
fsDefault	USHORT	
BYTE	unsigned char	b
CATCHBUF	struct	ctchbf
reserved[4]	ULONG	
CHARBUNDLE	struct	cbnd
lColor	LONG	
lBackColor	LONG	
usMixMode	USHORT	
usBackMixMode	USHORT	
usSet	USHORT	

APPENDIX B

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
usPrecision	USHORT	
sizfxCell	SIZEF	
ptlAngle	POINTL	
ptlShear	POINTL	
usDirection	USHORT	
CLASSINFO	struct	cls i
fClassStyle	ULONG	
pfnWindowProc	PFNWP	
cbWindowData	USHORT	
COLOR	LONG	clr
COMMANDMSG	struct	
source	USHORT	mp2
fMouse	BOOL	
cmd	USHORT	mp1
unused	USHORT	
COUNTRYCODE	struct	ctryc
country	USHORT	
codepage	USHORT	
COUNTRYINFO	struct	ctryi
country	USHORT	
codepage	USHORT	
fsDateFmt	USHORT	
szCurrency[5]	CHAR	
szThousandsSeparator[2]	CHAR	
szDecimal[2]	CHAR	
szDateSeparator[2]	CHAR	
szTimeSeparator[2]	CHAR	
fsCurrencyFmt	UCHAR	
cDecimalPlace	UCHAR	
fsTimeFmt	UCHAR	
abReserved1[2]	USHORT	Must be 0
szDataSeparator[2]	CHAR	
abReserved2[5]	USHORT	Must be 0
CREATESTRUCT	struct	crst
pPresParams	PVOID	
pCtlData	PVOID	
id	USHORT	
hwndInsertBehind	HWND	
hwndOwner	HWND	
cy	SHORT	
cx	SHORT	
y	SHORT	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
x	SHORT	
flStyle	ULONG	
pszText	PSZ	
pszClass	PSZ	
hwndParent	HWND	
CURSORINFO	struct	csri
hwnd	HWND	
x	SHORT	
y	SHORT	
cx	SHORT	
cy	SHORT	
fs	USHORT	
rcClip	RECTL	
DATETIME	struct	date
hours	UCHAR	
minutes	UCHAR	
seconds	UCHAR	
hundredths	UCHAR	
day	UCHAR	
month	UCHAR	
year	USHORT	
timezone	SHORT	
weekday	UCHAR	
DDEINIT	struct	ddei
cb	USHORT	
pszAppName	PSZ	
pszTopic	PSZ	
DDESTRICT	struct	dde
cbData	ULONG	
fsStatus	USHORT	
usFormat	USHORT	
offszItemName	USHORT	
offabData	USHORT	
DENA1	struct	
reserved	UCHAR	
cbName	UCHAR	
cbValue	USHORT	
szName[1]	UCHAR	
DEVOPENSTRUC	struct	dop
pszLogAddress	PSZ	
pszDriverName	PSZ	
pdriv	PDRIVDATA	

APPENDIX B

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
pszDataType	PSZ	
pszComment	PSZ	
pszQueueProcName	PSZ	
pszQueueProcParams	PSZ	
pszSpoolerParams	PSZ	
pszNetworkParams	PSZ	
DLGTEMPLATE	struct	dlgt
cbTemplate	USHORT	
type	USHORT	
codepage	USHORT	
offadlgti	USHORT	Must be 12
fsTemplateStatus	USHORT	
iItemFocus	USHORT	
coffPresParams	USHORT	Must be 0
adlgti[1]	DLGTITEM	
DLGTITEM	struct	dlgti
fsItemStatus	USHORT	
cChildren	USHORT	
cchClassName	USHORT	
offClassName	USHORT	
cchText	USHORT	
offText	USHORT	
flStyle	ULONG	
x	SHORT	
y	SHORT	
cx	SHORT	
cy	SHORT	
id	USHORT	
offPresParams	USHORT	
offCtlData	USHORT	
DOSFSRSEM	struct	dosfsrs
cb	USHORT	
pid	PID	
tid	TID	
cUsage	USHORT	
client	USHORT	
sem	ULONG	
DRIVDATA	struct	driv
cb	LONG	
IVersion	LONG	
szDeviceName[32]	CHAR	
abGeneralData[1]	CHAR	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
EAOP	struct	
fpGEAList	PGEALIST	
fpFEAList	PFEALIST	
oError	ULONG	
ENTRYFDATA	struct	efd
cb	USHORT	
cchEditLimit	USHORT	
ichMinSel	USHORT	
ichMaxSel	USHORT	
ERRINFO	struct	erri
cbFixedErrInfo	USHORT	
idError	ERRORID	
cDetailLevel	USHORT	
offaoffszMsg	USHORT	
offBinaryData	USHORT	
ERRORID	ULONG	errid
FATTRS	struct	
usRecordLength	USHORT	
fsSelection	USHORT	
lMatch	LONG	
szFacename[FACESIZE]	CHAR	
idRegistry	USHORT	
usCodePage	USHORT	
lMaxBaselineExt	LONG	
lAveCharWidth	LONG	
fsType	USHORT	
fsFontUse	USHORT	
FDATE	struct	fdate
unsigned day	: 5	
unsigned month	: 4	
unsigned year	: 7	
FEA	struct	
bRsvd	BYTE	
cbName	BYTE	
cbValue	USHORT	
FEALIST	struct	
cbList	ULONG	
list[1]	FEA	
FFDESCS[2][FACESIZE]	CHAR	ffdescs

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
FILEFINDBUF	struct	
fdateCreation	FDATE	
ftimeCreation	FTIME	
fdateLastAccess	FDATE	
ftimeLastAccess	FTIME	
fdateLastWrite	FDATE	
ftimeLastWrite	FTIME	
cbFile	ULONG	
cbFileAlloc	ULONG	
attrFile	USHORT	
cchName	UCHAR	
achName[13]	CHAR	
FILEFINDBUF2	struct	
fdateCreation	FDATE	
ftimeCreation	FTIME	
fdateLastAccess	FDATE	
ftimeLastAccess	FTIME	
fdateLastWrite	FDATE	
ftimeLastWrite	FTIME	
cbFile	ULONG	
cbFileAlloc	ULONG	
attrFile	USHORT	
cbList	ULONG	
cchName	UCHAR	
achName[256]	CHAR	
FILESTATUS	struct	fsts
fdateCreation	FDATE	
ftimeCreation	FTIME	
fdateLastAccess	FDATE	
ftimeLastAccess	FTIME	
fdateLastWrite	FDATE	
ftimeLastWrite	FTIME	
cbFile	ULONG	
cbFileAlloc	ULONG	
attrFile	USHORT	
FIXED	LONG	fx
FIXED88	USHORT	fx88
FIXED114	USHORT	fx114
FOCAFONT	struct	ff
fsSignature	FONTSIGNATURE	
fmMetrics	FOCAMETRICS	
fdDefinitions	FONTDEFINITIONHEADER	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
FOCAMETRICS	struct	foca
ulIdentity	ULONG	
ulSize	ULONG	
szFamilyname[32]	CHAR	
szFacename[32]	CHAR	
usRegistryId	SHORT	
usCodePage	SHORT	
yEmHeight	SHORT	
yXHeight	SHORT	
yMaxAscender	SHORT	
yMaxDescender	SHORT	
yLowerCaseAscent	SHORT	
yLowerCaseDescent	SHORT	
yInternalLeading	SHORT	
yExternalLeading	SHORT	
xAveCharWidth	SHORT	
xMaxCharInc	SHORT	
xEmInc	SHORT	
yMaxBaselineExt	SHORT	
sCharSlope	SHORT	
sInlineDir	SHORT	
sCharRot	SHORT	
usWeightClass	USHORT	
usWidthClass	USHORT	
xDeviceRes	SHORT	
yDeviceRes	SHORT	
usFirstChar	SHORT	
usLastChar	SHORT	
usDefaultChar	SHORT	
usBreakChar	SHORT	
usNominalPointSize	SHORT	
usMinimumPointSize	SHORT	
usMaximumPointSize	SHORT	
fsTypeFlags	SHORT	
fsDefn	SHORT	
fsSelectionFlags	SHORT	
fsCapabilities	SHORT	
ySubscriptXSize	SHORT	
ySubscriptYSize	SHORT	
ySubscriptXOffset	SHORT	
ySubscriptYOffset	SHORT	
ySuperscriptXSize	SHORT	
ySuperscriptYSize	SHORT	
ySuperscriptXOffset	SHORT	
ySuperscriptYOffset	SHORT	
yUnderscoreSize	SHORT	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
yUnderscorePosition	SHORT	
yStrikeoutSize	SHORT	
yStrikeoutPosition	SHORT	
usKerningPairs	SHORT	
usKerningTracks	SHORT	
pszDeviceNameOffset	PSZ	
FONTDEFINITIONHEADER	struct	fdh
ulIdentity	ULONG	
ulSize	ULONG	
fsFontdef	SHORT	
fsChardef	SHORT	
usCellSize	SHORT	
xCellWidth	SHORT	
yCellHeight	SHORT	
xCellIncrement	SHORT	
xCellA	SHORT	
xCellB	SHORT	
xCellC	SHORT	
pCellBaseOffset	SHORT	
FONTMETRICS	struct	fm
szFamilyname[FACESIZE]	CHAR	
szFacename[FACESIZE]	CHAR	
idRegistry	USHORT	
usCodePage	USHORT	
lEmHeight	LONG	
lXHeight	LONG	
lMaxAscender	LONG	
lMaxDescender	LONG	
lLowerCaseAscent	LONG	
lLowerCaseDescent	LONG	
lInternalLeading	LONG	
lExternalLeading	LONG	
lAveCharWidth	LONG	
lMaxCharInc	LONG	
lEmInc	LONG	
lMaxBaselineExt	LONG	
sCharSlope	SHORT	
sInlineDir	SHORT	
sCharRot	SHORT	
usWeightClass	USHORT	
usWidthClass	USHORT	
sXDeviceRes	SHORT	
sYDeviceRes	SHORT	
sFirstChar	SHORT	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
sLastChar	SHORT	
sDefaultChar	SHORT	
sBreakChar	SHORT	
sNominalPointSize	SHORT	
sMinimumPointSize	SHORT	
sMaximumPointSize	SHORT	
fsType	USHORT	
fsDefn	USHORT	
fsSelection	USHORT	
fsCapabilities	USHORT	
lSubscriptXSize	LONG	
lSubscriptYSize	LONG	
lSubscriptXOffset	LONG	
lSubscriptYOffset	LONG	
lSuperscriptXSize	LONG	
lSuperscriptYSize	LONG	
lSuperscriptXOffset	LONG	
lSuperscriptYOffset	LONG	
lUnderscoreSize	LONG	
lUnderscorePosition	LONG	
lStrikeoutSize	LONG	
lStrikeoutPosition	LONG	
sKerningPairs	SHORT	
sReserved	SHORT	
lMatch	LONG	
FontSignature	struct	fs
ulIdentity	ULONG	
ulSize	ULONG	
achSignature[12]	CHAR	
FrameCDATA	struct	fcdata
cb	USHORT	
flCreateFlags	ULONG	
hmodResources	HMODULE	
idResources	USHORT	
FSALLOCATE	struct	fsalloc
idFileSystem	ULONG	
cSectorUnit	ULONG	
cUnit	ULONG	
cUnitAvail	ULONG	
cbSector	USHORT	
FTIME	struct	ftime
unsigned twosecs	: 5	
unsigned minutes	: 6	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
unsigned hours	: 5	
GEA	struct	
cbName	BYTE	
szName[1]	CHAR	
GEALIST	struct	
cbList	ULONG	
list[1]	GEA	
GINFOSEG	struct	gis
time	ULONG	
msecs	ULONG	
hour	UCHAR	
minutes	UCHAR	
seconds	UCHAR	
hundredths	UCHAR	
timezone	USHORT	
cusecTimerInterval	USHORT	
day	UCHAR	
month	UCHAR	
year	USHORT	
weekday	UCHAR	
uchMajorVersion	UCHAR	
uchMinorVersion	UCHAR	
chRevisionLetter	UCHAR	
sgCurrent	UCHAR	
sgMax	UCHAR	
cHugeShift	UCHAR	
fProtectModeOnly	UCHAR	
pidForeground	USHORT	
fDynamicSched	UCHAR	
csecMaxWait	UCHAR	
cmsecMinSlice	USHORT	
cmsecMaxSlice	USHORT	
bootdrive	USHORT	
amecRAS[32]	UCHAR	
csgWindowableVioMax	UCHAR	
csgPMMMax	UCHAR	
GRADIENTL	struct	gradl
x	LONG	
y	LONG	
HAB	LHANDLE	hab
HACCEL	LHANDLE	hacce1
HAPP	LHANDLE	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
HATOMTBL	LHANDLE	
HBITMAP	LHANDLE	hbm
HCINFO	struct	hci
szFormname[32]	CHAR	
cx	LONG	
cy	LONG	
xLeftClip	LONG	
yBottomClip	LONG	
xRightClip	LONG	
yTopClip	LONG	
xPels	LONG	
yPels	LONG	
flAttributes	LONG	
HDC	LHANDLE	hdc
HDIR	SHANDLE	hdir
HENUM	LHANDLE	henum
HFILE	SHANDLE	hf
HHEAP	LHANDLE	
HKBD	SHANDLE	
HMF	LHANDLE	hmf
HMODULE	USHORT	hmod
HMONITOR	SHANDLE	hmon
HMOU	SHANDLE	
HMQ	LHANDLE	hmq
HPIPE	SHANDLE	hp
HPOINTER	LHANDLE	hptr
HPROC	LHANDLE	hproc
HPROGARRAY	struct	hpga
ahprog[1]	HPROGRAM	
HPROGRAM	LHANDLE	hprog
HPS	LHANDLE	hps
HQUEUE	SHANDLE	hq
HRGN	LHANDLE	hrgn
HSEM	VOID FAR *	hsem

APPENDIX B

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
HSPL	LHANDLE	hspl
HSTD	LHANDLE	hstd
HSWITCH	LHANDLE	hsw
HSYSSEM	LHANDLE	hssm
HTIMER	SHANDLE	
HVIO	SHANDLE	
HVPS	USHORT	hpvs
HWND	LHANDLE	hwnd
IMAGEBUNDLE	struct	ibmd
lColor	LONG	
lBackColor	LONG	
usMixMode	USHORT	
usBackMixMode	USHORT	
KbdHWID	struct	
length	USHORT	
kbdLid	USHORT	
reserved1	USHORT	
reserved2	USHORT	
KBDINFO	struct	kbst
cb	USHORT	
fsMask	USHORT	
chTurnAround	USHORT	
fsInterim	USHORT	
fsState	USHORT	
KBDKEYINFO	struct	kbci
chChar	UCHAR	
chScan	UCHAR	
fbStatus	UCHAR	
bNlsShift	UCHAR	*** Must be 0 ***
fsState	USHORT	
time	ULONG	
KBDTRANS	struct	kbtr
chChar	UCHAR	
chScan	UCHAR	
fbStatus	UCHAR	
bNlsShift	UCHAR	Must be 0
fsState	USHORT	
time	ULONG	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
fsDD	USHORT	
fsXlate	USHORT	
fsShift	USHORT	
sZero	USHORT	
KERNINGPAIRS	struct	knpnr
sFirstChar	SHORT	
sSecondChar	SHORT	
sKerningAmount	SHORT	
LHANDLE	void far *	
LINEBUNDLE	struct	lbnd
lColor	LONG	
lReserved	LONG	
usMixMode	USHORT	
usReserved	USHORT	
fxWidth	FIXED	
lGeomWidth	LONG	
usType	USHORT	
usEnd	USHORT	
usJoin	USHORT	
LINFOSEG	struct	lis
pidCurrent	PID	
pidParent	PID	
prtyCurrent	USHORT	
tidCurrent	TID	
sgCurrent	USHORT	
rfProcStatus	UCHAR	
dummy1	UCHAR	
ffForeground	BOOL	
typeProcess	UCHAR	
dummy2	UCHAR	
selEnvironment	SEL	
offCmdLine	USHORT	
cbDataSegment	USHORT	
cbStack	USHORT	
cbHeap	USHORT	
hmod	HMODULE	
selDS	SEL	
LORDER	struct	lord
idCode	UCHAR	
uchLength	UCHAR	
uchData[LORDER_ML]	UCHAR	

APPENDIX B

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
MARKERBUNDLE	struct	mbnd
lColor	LONG	
lBackColor	LONG	
usMixMode	USHORT	
usBackMixMode	USHORT	
usSet	USHORT	
usSymbol	USHORT	
sizfxCell	SIZEF	
MATRIXLF	struct	matlf
fxM11	FIXED	
fxM12	FIXED	
lM13	LONG	
fxM21	FIXED	
fxM22	FIXED	
lM23	LONG	
lM31	LONG	
lM32	LONG	
lM33	LONG	
MENUITEM	struct	mi
iPosition	SHORT	
afStyle	USHORT	
afAttribute	USHORT	
id	USHORT	
hwndSubMenu	HWND	
hItem	ULONG	
MOUEVENTINFO	struct	mouev
fs	USHORT	
time	ULONG	
row	USHORT	
col	USHORT	
MOUQUEINFO	struct	mouqi
cEvents	USHORT	
cmaxEvents	USHORT	
MPARAM	VOID FAR *	mp
MQINFO	struct	mqi
cb	USHORT	
pid	PID	
tid	TID	
cmsgs	USHORT	
pReserved	PVOID	
MRESULT	VOID FAR *	mres

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
MUXSEM	struct	mxs
zero	USHORT	
hsem	HSEM	
MUXSEMLIST	struct	mxsl
cmxs	USHORT	
amxs[16]	MUXSEM	
NOPTRRECT	struct	mourt
row	USHORT	
col	USHORT	
cRow	USHORT	
cCol	USHORT	
NPBYTE	BYTE near *	
NPCH	char near *	
NPPOINTL	POINTL near *	
NPRECTL	RECTL near *	
NPRESPARAMS	PRESPARAMS near *	
NPSZ	char near *	
NPWPOINT	WPOINT near *	
NPWRECT	WRECT near *	
ODPOINT	struct	odpt
dx	CHAR	
dy	CHAR	
ORDER	struct	ord
idCode	UCHAR	
uchData	UCHAR	
ORDER_GBEL	struct	ogbel
lElementType	LONG	
achDesc[GBEL_DL]	CHAR	
ORDER_GBPTH	struct	ogbpth
usReserved	USHORT	
idPath	LONG	
ORDER_GCALLS	struct	ogcalls
sReserved	USHORT	
idSegment	LONG	
ORDER_GCBIMG	struct	ogbimg
uchFormat	UCHAR	
uchReserved	UCHAR	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
cx	SWPUSHORT	
cy	SWPUSHORT	
ORDER_GEESCP	struct	ogeescp
uchType	UCHAR	
uchIdent	UCHAR	
auchData[GEESCP_ML]	UCHAR	
ORDER_GESCP	struct	ogescp
uchType	UCHAR	
uchIdent	UCHAR	
auchData[GESCP_ML]	UCHAR	
ORDER_GFPPTH	struct	ogfpth
fbFlags	UCHAR	
uchReserved	UCHAR	
idPath	LONG	
ORDER_GMPPTH	struct	ogmpth
uchMode	UCHAR	
uchReserved	UCHAR	
idPath	LONG	
ORDER_GSBICOL	struct	ogbicol
fbFlags	UCHAR	
auchColor[3]	UCHAR	
ORDER_GSCPTH	struct	ogscpth
fbFlags	UCHAR	
uchReserved	UCHAR	
idPath	LONG	
ORDER_GSGCH	struct	ogsgch
uchIdent	UCHAR	
auchData[GSGCH_ML]	UCHAR	
ORDER_GSIA	struct	ogsia
uchAttrType	UCHAR	
uchPrimType	UCHAR	
fbFlags	UCHAR	
auchValue[GSIA_VL]	UCHAR	
ORDERL_GBBLT	struct	olgbblt
fsFlags	USHORT	
usMix	USHORT	
hbmSrc	HBITMAP	
lOptions	LONG	
rcfTargetRect	RECTL	
rcfSourceRect	RECTL	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
ORDERL_GCARC ptInter ptEnd	struct POINTL POINTL	olgcarc
ORDERL_GCBOX fbFlags uchReserved ptCorner hAxis vAxis	struct UCHAR UCHAR POINTL LONG LONG	olgcbox
ORDERL_GCCHSTE fbFlags uchReserved ptRect[2] cchString achString[1] adx[1]	struct UCHAR UCHAR POINTL SWPUSHORT CHAR LONG	olgcchste
ORDERL_GCPARC ptCenter ufxMultiplier usStartAngle usSweepAngle	struct POINTL FIXED LONG LONG	olgcparc
ORDERL_GCSFLT apt[2*GCSFLT_SMF] afxSharpness[GCSFLT_SMF]	struct POINTL FIXED	olgcsflt
ORDERL_GEBB fbFlags usMix cPoints hbmSrc lReserved lOptions aptPoints[GEBB_LMP]	struct UCHAR USHORT UCHAR HBITMAP LONG LONG POINTL	olgebb
ORDERL_GSAP p q r s	struct LONG LONG LONG LONG	olgsap
ORDERL_GSCC cxInt cyInt cxFract cyFract	struct LONG LONG USHORT USHORT	olgsc

APPENDIX B

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
fbFlags	UCHAR	
uchReserved	UCHAR	
ORDERL_GSMC	struct	olgsmc
cx	LONG	
cy	LONG	
fbFlags	UCHAR	
uchReserved	UCHAR	
ORDERL_GSPRP	struct	olgsprp
fbFlags	UCHAR	
uchReserved	UCHAR	
ptPos	POINTL	
ORDERL_GSSB	struct	olgssb
fbFlags	UCHAR	
fbMask	UCHAR	
alMatrix[GSSB_ML]	LONG	
ORDERL_GSSLW	struct	olgsslw
fbFlags	UCHAR	
uchReserved	UCHAR	
LineWidth	LONG	
ORDERL_GSTM	struct	olgstm
uchReserved	UCHAR	
fbFlags	UCHAR	
fsMask	USHORT	
alMatrix[GSTM_ML]	LONG	
ORDERS_GBBLT	struct	osgblt
fsFlags	USHORT	
usMix	USHORT	
hbmSrc	HBITMAP	
lOptions	LONG	
rcsTargetRect	RECT1S	
rcsSourceRect	RECTL	
ORDERS_GCARC	struct	osgcarc
ptInter	POINTS	
ptEnd	POINTS	
ORDERS_GCBOX	struct	osgcbbox
fbFlags	UCHAR	
uchReserved	UCHAR	
ptCorner	POINTS	
hAxis	SHORT	
vAxis	SHORT	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
ORDERS_GCCHSTE	struct	osgchste
fbFlags	UCHAR	
uchReserved	UCHAR	
ptRect[2]	POINTS	
cchString	SWPUSHORT	
achString[1]	CHAR	
adx[1]	SHORT	
ORDERS_GCPARC	struct	osgcparc
ptCenter	POINTS	
ufx88Multiplier	FIXED88	
usStartAngle	LONG	
usSweepAngle	LONG	
ORDERS_GCSFLT	struct	osgcsflt
apt[2*GCSFLT_SMF]	POINTS	
afxSharpness[GCSFLT_SMF]	FIXED	
ORDERS_GSAP	struct	osgsap
p	SHORT	
q	SHORT	
r	SHORT	
s	SHORT	
ORDERS_GSCC	struct	osgsc
cxInt	SHORT	
cyInt	SHORT	
cxFract	USHORT	
cyFract	USHORT	
fbFlags	UCHAR	
uchReserved	UCHAR	
ORDERS_GSMC	struct	osgsmc
cx	SHORT	
cy	SHORT	
fbFlags	UCHAR	
uchReserved	UCHAR	
ORDERS_GSPRP	struct	osgsprp
fbFlags	UCHAR	
uchReserved	UCHAR	
ptPos	POINTS	
ORDERS_GSSB	struct	osgssb
fbFlags	UCHAR	
fbMask	UCHAR	
alMatrix[GSSB_ML]	SHORT	

APPENDIX B

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
ORDERS_GSSLW	struct	osgsslw
fbFlags	UCHAR	
uchReserved	UCHAR	
LineWidth	SHORT	
ORDERS_GSTM	struct	osgstm
uchReserved	UCHAR	
fbFlags	UCHAR	
fsMask	USHORT	
asMatrix[GSTM_ML]	SHORT	
OWNERITEM	struct	oi
hwnd	HWND	
hps	HPS	
fsState	USHORT	
fsAttribute	USHORT	
fsStateOld	USHORT	
fsAttributeOld	USHORT	
rcItem	RECTL	} *** This field contains idItem for menus, iItem for listboxes. ***
idItem	SHORT	
hItem	ULONG	
PACCEL	ACCEL FAR *	
PACCELTABLE	ACCELTABLE FAR *	
PARCPARAMS	ARCPARAMS FAR *	
PBITMAPFILEHEADER	BITMAPFILEHEADER FAR *	
PBITMAPINFO	BITMAPINFO FAR *	
PBITMAPINFOHEADER	BITMAPINFOHEADER FAR *	
PBOOL	BOOL FAR *	
PBTNCDATA	BTNCDATA FAR *	
PBUNDLE	PVOID	
PBYTE	BYTE FAR *	
PCATCHBUF	CATCHBUF FAR *	
PCH	char far *	
PCHAR	CHAR FAR *	
PCLASSINFO	CLASSINFO FAR *	
PCOLOR	COLOR FAR *	
PCOUNTRYCODE	COUNTRYCODE FAR *	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
PCOUNTRYINFO	COUNTRYINFO FAR *	
PCREATESTRUCT	CREATESTRUCT FAR *	
PCURSORINFO	CURSORINFO FAR *	
PDATETIME	DATETIME FAR *	
PDDEINIT	DDEINIT FAR *	
PDDESTRUCT	DDESTRUCT FAR *	
PDEVOPENDATA	PSZ FAR *	
PDEVOPENSTRUC	DEVOPENSTRUC FAR *	
PDLGTEMPLATE	DLGTEMPLATE FAR *	
PDLGTITEM	DLGTITEM FAR *	
PDOSFSRSEM	DOSFSRSEM FAR *	
PDRIVDATA	DRIVDATA far *	
PEAOP	EAOP FAR *	
PENTRYFDATA	ENTRYFDATA FAR *	
PERRINFO	ERRINFO FAR *	
PERRORID	ERRORID FAR *	
PFATTRS	FATTRS far *	
PFDATE	FDATE FAR *	
PFEA	FEA FAR *	
PFEALIST	FEALIST FAR *	
PFFDESCS	FFDESCS FAR *	
PFILEFINDBUF	FILEFINDBUF FAR *	
PFILEFINDBUF2	FILEFINDBUF2 FAR *	
PFILESTATUS	FILESTATUS FAR *	
PFIXED	FIXED FAR *	
PFOCAFONT	FOCAFONT FAR *	
PFOCAMETRICS	FOCAMETRICS FAR *	
PFONTDEFINITIONHEADER	FONTDEFINITIONHEADER FAR *	
PFONTMETRICS	FONTMETRICS far *	
PONTSIGNATURE	ONTSIGNATURE FAR *	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
PFRAMECDATA	FRAMECDATA FAR *	
PFSALLOCATE	FSALLOCATE FAR *	
PFTIME	FTIME FAR *	
PGEA	GEA FAR *	
PGEALIST	PGEALIST FAR *	
PGINFOSEG	GINFOSEG FAR *	
PGRADIENTL	GRADIENTL FAR *	
PHAB	HAB FAR *	
PHBITMAP	HBITMAP FAR *	
PHCINFO	HCINFO FAR *	
PHDC	HDC FAR *	
PHDIR	HDIR FAR *	
PHFILE	HFILE FAR *	
PHKBD	HKBD far *	
PHMF	HMF FAR *	
PHMODULE	HMODULE FAR *	
PHMONITOR	HMONITOR FAR *	
PHMOU	HMOU far *	
PHPIPE	HPIPE FAR *	
PHPROGARRAY	HPROGARRAY FAR *	
PHPROGRAM	HPROGRAM FAR *	
PHPS	HPS FAR *	
PHQUEUE	HQUEUE FAR *	
PHRGN	HRGN FAR *	
PHSEM	HSEM FAR *	
PHSWITCH	HSWITCH FAR *	
PHSYSSEM	HSYSSEM FAR *	
PHTIMER	HTIMER FAR *	
PHVIO	HVIO far *	
PHVPS	HVPS far *	phpvs

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
PHWND	HWND FAR *	
PIBBUFFER	struct	pibuf
progt	PROGTYPE	
szTitle[MAXNAMEL+1]	CHAR	
szIconFileName[MAXPATHL+1]	CHAR	
szExecutable[MAXPATHL+1]	CHAR	
szStartupDir[MAXPATHL+1]	CHAR	
xywinInitial	XYWINSIZE	
res1	USHORT	
res2	LHANDLE	
cchEnvironmentVars	USHORT	
pchEnvironmentVars	PCH	
cchProgramParameter	USHORT	
pchProgramParameter	PCH	
HEPstrings[1024]	CHAR	
PIBSTRUCT	struct	pib
progt	PROGTYPE	
szTitle[MAXNAMEL+1]	CHAR	
szIconFileName[MAXPATHL+1]	CHAR	
szExecutable[MAXPATHL+1]	CHAR	
szStartupDir[MAXPATHL+1]	CHAR	
xywinInitial	XYWINSIZE	
res1	USHORT	
res2	LHANDLE	
cchEnvironmentVars	USHORT	
pchEnvironmentVars	PCH	
cchProgramParameter	USHORT	
pchProgramParameter	PCH	
PID	USHORT	pid
PIDINFO	struct	pidi
pid	PID	
tid	TID	
pidParent	PID	
PINT	INT FAR *	
PKbdHWID	KbdHWID FAR *	
PKBDINFO	KBDINFO far *	
PKBDKEYINFO	KBDKEYINFO far *	
PKBDTRANS	KBDTRANS far *	
PKERNINGPAIRS	KERNINGPAIRS FAR *	
PLINFOSEG	LINFOSEG FAR *	

APPENDIX B

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
PLONG	LONG FAR *	
PMATRIXLF	MATRIXLF FAR *	
PMENUITEM	MENUITEM FAR *	
PMOUEVENTINFO	MOUEVENTINFO far *	
PMOUCUEINFO	MOUCUEINFO far *	
PMPARAM	MPARAM FAR *	pmp
PMQINFO	MQINFO FAR *	
PMRESULT	MRESULT FAR *	pmres
PMUXSEM	MUXSEM FAR *	
PMUXSEMLIST	MUXSEMLIST FAR *	
PNOPTRRECT	NOPTRRECT far *	
POINTERINFO	struct	ptri
fPointer	BOOL	
xHotspot	SHORT	
yHotspot	SHORT	
hbmPointer	HBITMAP	
hbmColor	HBITMAP	
POINTL	struct	ptl
x	LONG	
y	LONG	
POINTS	struct	pts
x	SHORT	
y	SHORT	
POWNERITEM	OWNERITEM FAR *	
PIBBUFFER	PIBBUFFER far *	
PIBSTRUCT	PIBSTRUCT FAR *	
PPID	PID FAR *	
PPIDINFO	PIDINFO FAR *	
PPOINTERINFO	POINTERINFO FAR *	
PPOINTL	POINTL FAR *	
PPOINTS	POINTS FAR *;	
PPRESPARAMS	PRESPARAMS FAR *	
PRFPROFILE	PRFPROFILE FAR *	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
PPROGRAMENTRY	PROGRAMENTRY FAR *	
PPROGTITLE	PROGTITLE FAR *	
PPROGTYPE	PROGTYPE FAR *	
PPTRLOC	PTRLOC far *	
PPTRSHAPE	PTRSHAPE far *	
PQFEOUTBLK	QFEOUTBLK FAR *	
PQMOPENDATA	PSZ FAR *	pqmdop
PQMSG	QMSG FAR *	
PQPOPENDATA	PSZ FAR *	pqpdp
PRECTL	RECTL FAR *	
PRESPARAMS	struct	pres
cb	ULONG	
aparam[1]	PARAM	
PRESULTCODES	RESULTCODES FAR *	
PRFPROFILE	struct	prfpro
cchUserName	ULONG	
pszUserName	PSZ	
cchSysName	ULONG	
pszSysName	PSZ	
PRGNRECT	RGNRECT FAR *	
PROGCATEGORY	CHAR	progc
PROGDETAILS	struct	progde
Length	ULONG	
progt	PROGTYPE	
pad1[3]	USHORT	
pszTitle	PSZ	
pszExecutable	PSZ	
pszParameters	PSZ	
pszStartupDir	PSZ	
pszIcon	PSZ	
pszEnvironment	PSZ	
swpInitial	SWP	
pad2[5]	USHORT	
PROGRAMENTRY	struct	proge
hprog	HPROGRAM	
progt	PROGTYPE	
szTitle[MAXNAMEL+1]	CHAR	

APPENDIX B

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
PROGTITLE	struct	
hprog	HPROGRAM	progti
progt	PROGTYPE	
pad1[3]	USHORT	
pszTitle	PSZ	
PROGTYPE	struct	
progc	PROGCATEGORY	progt
fbVisible	UCHAR	
PSBCDATA	SBCDATA FAR *	
PSCALEFACT	SCALEFACT far *	
PSEL	SEL FAR *	
PSHORT	SHORT FAR *	
PSIZEF	SIZEF FAR *	
PSIZEL	SIZEL FAR *	
PSMHSTRUCT	SMHSTRUCT FAR *	
PSTARTDATA	STARTDATA FAR *	
PSTATUSDATA	STATUSDATA FAR *	
PSTR8	STR8 FAR *	
PSTR16	STR16 FAR *	
PSTR32	STR32 FAR *	
PSTR64	STR64 FAR *	
PSTRINGINBUF	STRINGINBUF far *	
PSWBLOCK	SWBLOCK FAR *	
PSWCNTRL	SWCNTRL FAR *	
PSWENTRY	SWENTRY FAR *	
PSWP	SWP FAR *	
PSZ	char far *	
PTID	TID FAR *	
PTRACKINFO	TRACKINFO FAR *	
PTRLOC	struct	
row	USHORT	moup1
col	USHORT	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
PTRSHAPE	struct	moups
cb	USHORT	
col	USHORT	
row	USHORT	
colHot	USHORT	
rowHot	USHORT	
PUCHAR	UCHAR FAR *	
PUINT	UINT FAR *	
PULONG	ULONG FAR *	
PUSERBUTTON	USERBUTTON FAR *;	
PUSHORT	USHORT FAR *	
PVIColorReg	VIColorReg FAR *	
PVIOCONFIGINFO	VIOCONFIGINFO far *	
PVIOCURLINFO	VIOCURLINFO FAR *	
PVIOFONTINFO	VIOFONTINFO far *	
PVIOINTENSITY	VIOINTENSITY far *	
PVIOMODEINFO	VIMODEINFO FAR *	
PVIOOVERSCAN	VIOOVERSCAN far *	
PVIOPALSTATE	VIOPALSTATE far *	
PVIOPHYSBUF	VIOPHYSBUF far *	
PVIOSETTARGET	VIOSETTARGET FAR *	
PVIOSETULINELOC	VIOSETULINELOC FAR *	
PVOID	VOID FAR *	
PWNDPARAMS	WNDPARAMS FAR *	
PWPOINT	WPOINT FAR *	
PWRECT	WRECT FAR *	
PXYWINSIZE	XYWINSIZE FAR *	
QFEOUTBLK	struct	qfeout
Total	USHORT	
Count	USHORT	
ProgramArr[1]	HPROGRAM	

APPENDIX B

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
QMSG	struct	qmsg
hwnd	HWND	
msg	USHORT	
mp1	MPARAM	
mp2	MPARAM	
time	ULONG	
ptl	POINTL	
QVERSDATA	struct	qver
environment	USHORT	
version	USHORT	
RECTIS	struct	rcs
xLeft	SHORT	
yBottom	SHORT	
xRight	SHORT	
yTop	SHORT	
RECTL	struct	rc1
xLeft	LONG	
yBottom	LONG	
xRight	LONG	
yTop	LONG	
RESULTCODES	struct	resc
codeTerminate	USHORT	
codeResult	USHORT	
RGB	struct	rgb
bBlue	BYTE	
bGreen	BYTE	
bRed	BYTE	
RGNRECT	struct	rgnrc
ircStart	USHORT	
crc	USHORT	
crcReturned	USHORT	
usDirection	USHORT	
SBCDATA	struct	sbcd
cb	USHORT	
sHilite	USHORT	*** Must be 0 ***
posFirst	SHORT	
posLast	SHORT	
posThumb	SHORT	
SCALEFACT	struct	mousc
rowScale	USHORT	
colScale	USHORT	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
SEL	unsigned short	sel
SHANDLE	unsigned short	
SIZEF	struct	sizfx
cx	FIXED	
cy	FIXED	
SIZEL	struct	sizl
cx	LONG	
cy	LONG	
SIZES	struct	sizs
cx	SHORT	
cy	SHORT	
SMHSTRUCT	struct	smhs
mp2	MPARAM	
mp1	MPARAM	
msg	USHORT	
hwnd	HWND	
STARTDATA	struct	stdata
Length	USHORT	
Related	USHORT	
FgBg	USHORT	
TraceOpt	USHORT	
PgmTitle	PSZ	
PgmName	PSZ	
PgmInputs	PBYTE	
TermQ	PBYTE	
Environment	PBYTE	
InheritOpt	USHORT	
SessionType	USHORT	
IconFile	PSZ	
PgmHandle	ULONG	
PgmControl	USHORT	
InitXPos	USHORT	
InitYPos	USHORT	
InitXSize	USHORT	
InitYSize	USHORT	
STATUSDATA	struct	stsdta
Length	USHORT	
SelectInd	USHORT	
BondInd	USHORT	
STR8[8]	CHAR	str8
STR16[16]	CHAR	str16

APPENDIX B

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
STR32[32]	CHAR	str32
STR64[64]	CHAR	str64
STRINGINBUF	struct	kbsi
cb	USHORT	
cchIn	USHORT	
SWBLOCK	struct	swblk
cswentry	USHORT	
aswentry[1]	SWENTRY	
SWCNTRL	struct	swctl
hwnd	HWND	
hwndIcon	HWND	
hprog	HPROGRAM	
idProcess	USHORT	
idSession	USHORT	
uchVisibility	UCHAR	
fbJump	UCHAR	
szSwtitle[MAXNAMEL+1]	CHAR	
fReserved	BYTE	
SWENTRY	struct	swent
hswitch	HSWITCH	
swctl	SWCNTRL	
SWP	struct	swp
fs	USHORT	
cy	SHORT	
cx	SHORT	
y	SHORT	
x	SHORT	
hwndInsertBehind	HWND	
hwnd	HWND	
SWPUSHORT	struct	swpus
HiByte	UCHAR	
LoByte	UCHAR	
TID	USHORT	tid
TRACKINFO	struct	ti
cxBorder	SHORT	
cyBorder	SHORT	
cxGrid	SHORT	
cyGrid	SHORT	
cxKeyboard	SHORT	
cyKeyboard	SHORT	
rcfTrack	RECTL	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
rcBoundary	RECTL	
ptlMinTrackSize	POINTL	
ptlMaxTrackSize	POINTL	
fs	USHORT	
UCHAR	unsigned char	uch
UINT	unsigned int	ui
ULONG	unsigned long	ul
USERBUTTON	struct	ubtn
hwnd	HWND	
hps	HPS	
fsState	USHORT	
fsStateOld	USHORT	
USHORT	unsigned short	us
VIOCOLORREG	struct	viocreg
cb	USHORT	
type	USHORT	
firstcolorreg	USHORT	
numcolorregs	USHORT	
colorregaddr	PCH	
VIOCONFIGINFO	struct	vioin
cb	USHORT	
adapter	USHORT	
display	USHORT	
cbMemory	ULONG	
Configuration	USHORT	
VDHVersion	USHORT	
Flags	USHORT	
HWBufferSize	ULONG	
FullSaveSize	ULONG	
PartSaveSize	ULONG	
EMAdaptersOFF	USHORT	
EMDisplaysOFF	USHORT	
VIOCURSORINFO	struct	vioci
yStart	USHORT	
cEnd	USHORT	
cx	USHORT	
attr	USHORT	
VIOFONTINFO	struct	viofi
cb	USHORT	
type	USHORT	
cxCell	USHORT	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
cyCell	USHORT	
pbData	PVOID	
cbData	USHORT	
VIOINTENSITY	struct	vioint
cb	USHORT	
type	USHORT	
fs	USHORT	
VIOMODEINFO	struct	viomi
cb	USHORT	
fbType	UCHAR	
color	UCHAR	
col	USHORT	
row	USHORT	
hres	USHORT	
vres	USHORT	
fmt_ID	UCHAR	
attrib	UCHAR	
VIOOVERSCAN	struct	vioos
cb	USHORT	
type	USHORT	
color	USHORT	
VIOPALSTATE	struct	viopal
cb	USHORT	
type	USHORT	
iFirst	USHORT	
acolor[1]	USHORT	
VIOPHYSBUF	struct	viopb
pBuf	PBYTE	
cb	ULONG	
asel[1]	SEL	
VIOSETTARGET	struct	viosett
cb	USHORT	
type	USHORT	
defaultalgorithm	USHORT	
VIOSETULINELOC	struct	viouline
cb	USHORT	
type	USHORT	
scanline	USHORT	
VORDER	struct	vord
idCode	UCHAR	
uchQualifier	UCHAR	

OS/2 Data Type or Structure Name	Component Type(s)	Variable Prefix
uchLength	SWPUSHORT	
uchData[VORDER_ML]	UCHAR	
WNDPARAMS	struct	wprm
fsStatus	USHORT	
cchText	USHORT	
pszText	PSZ	
cbPresParams	USHORT	
pPresParams	PVOID	
cbCtlData	USHORT	
pCtlData	PVOID	
WPOINT	struct	wpt
x	SHORT	
dummy1	SHORT	
y	SHORT	
dummy2	SHORT	
WRECT	struct	wrc
xLeft	SHORT	
dummy1	SHORT	
yBottom	SHORT	
dummy2	SHORT	
xRight	SHORT	
dummy3	SHORT	
yTop	SHORT	
dummy4	SHORT	
XYWINSIZE	struct	xywin
x	SHORT	
y	SHORT	
cx	SHORT	
cy	SHORT	
fsWindow	USHORT	

C

Variable Prefixes

As you have seen throughout this book, the use of the suggested naming conventions for program variables will make your source code more readable and consistent with the OS/2 data types and function definitions.

This appendix contains a list of the OS/2 defined data types and the suggested prefix for naming variables of this type.

To see the definitions of these data types, please refer to Appendix B.

Prefix	OS/2 Data Type
acc	ACCEL
acct	ACCELTABLE
arcp	ARCPARAMS
b	BYTE
bfh	BITMAPFILEHEADER
bmi	BITMAPINFO
bmp	BITMAPINFOHEADER
btncd	BTNCDATA
cbnd	CHARBUNDLE
clr	COLOR
clsi	CLASSINFO
crst	CREATESTRUCT
csri	CURSORINFO
ctchbf	CATCHBUF
ctryc	COUNTRYCODE
ctryi	COUNTRYINFO
date	DATETIME

Prefix	OS/2 Data Type
dde	DDESTRUCT
ddei	DDEINIT
dlgt	DLGTEMPLATE
dlgti	DLGTITEM
dop	DEVOPENSTRUC
dosfsrs	DOSFSRSEM
driv	DRIVDATA
efd	ENTRYFDATA
erri	ERRINFO
errid	ERRORID
f	BOOL
fcdata	FRAMECDATA
fdate	FDATE
fdh	FONTDEFINITIONHEADER
ff	FOCAFONT
ffdescs	FFDESCS[2][FACESIZE]
fm	FONTMETRICS
foca	FOCAMETRICS
fs	FONTSIGNATURE
fsalloc	FSALLOCATE
fsts	FILESTATUS
ftime	FTIMEy
fx	FIXED
fx88	FIXED88
fx114	FIXED114
gis	GINFOSEG
gradl	GRADIENTL
hab	HAB
haccel	HACCEL
hbm	HBITMAP
hci	HCINFO
hdc	HDC
hdir	HDIR
henum	HENUM
hf	HFILE
hmf	HMF
hmod	HMODULE
hmon	HMONITOR
hmq	HMQ
hp	HPIPE
hpga	HPROGARRAY
hproc	HPROC
hprog	HPROGRAM
hps	HPS
hptr	HPOINTER
hpvs	HVPS
hq	HQUEUE

Prefix	OS/2 Data Type
hrgn	HRGN
hsem	HSEM
hspl	HSPL
hssm	HSYSSEM
hstd	HSTD
hsw	HSWITCH
hwnd	HWND
ibmd	IMAGEBUNDLE
kbc i	KBDKEYINFO
kbsi	STRINGINBUF
kbst	KBDINFO
kbxl	KBDXLATE
krnpr	KERNINGPAIRS
lbnd	LINEBUNDLE
lis	LINFOSEG
lord	LORDER
matlf	MATRIXLF
mbnd	MARKERBUNDLE
mi	MENUIITEM
mouev	MOUEVENTINFO
moupl	PTRLOC
moups	PTRSHAPE
mouqi	MOUQUEINFO
mourt	NOPTRRECT
mousc	SCALEFACT
mp	MPARAM
mpl	MPARAM (number 1) (for messages)
mp2	MPARAM (number 2) (for messages)
mqi	MQINFO
mres	MRESULT
mxs	MUXSEM
mxsl	MUXSEMLIST
odpt	ODPOINT
ogbel	ORDER_GBEL
ogbicol	ORDER_GSBICOL
ogbimg	ORDER_GCBIMG
ogbpth	ORDER_GBPTH
ogcalls	ORDER_GCALLS
ogeescp	ORDER_GEESCP
ogescp	ORDER_GESCP
ogfpth	ORDER_GFPTH
ogmpth	ORDER_GMPTH
ogscpth	ORDER_GSCPTH
ogsgch	ORDER_GSGCH
ogsia	ORDER_GSIA
oi	OWNERITEM
olgbblt	ORDERL_GBBLT

Prefix	OS/2 Data Type
olgcarc	ORDERL_GCARC
olgcbbox	ORDERL_GCBOX
olgcchste	ORDERL_GCCHSTE
olgcparc	ORDERL_GCPARC
olgcsflt	ORDERL_GCSFLT
olgebb	ORDERL_GEBB
olgsap	ORDERL_GSAP
olgscc	ORDERL_GSCC
olgsmc	ORDERL_GSMC
olgsprp	ORDERL_GSPRP
olgssb	ORDERL_GSSB
olgsslw	ORDERL_GSSLW
olgstm	ORDERL_GSTM
ord	ORDER
osgbblt	ORDERS_GBBLT
osgcarc	ORDERS_GCARC
osgcbbox	ORDERS_GCBOX
osgchste	ORDERS_GCCHSTE
osgparc	ORDERS_GCPARC
osgcsflt	ORDERS_GCSFLT
oggsap	ORDERS_GSAP
oggscc	ORDERS_GSCC
oggsmc	ORDERS_GSMC
oggsprp	ORDERS_GSPRP
oggsb	ORDERS_GSSB
ogsslw	ORDERS_GSSLW
ogstm	ORDERS_GSTM
pbnd	AREABUNDLE
phvps	PHVPS
pib	PIBSTRUCT
pibuf	PIBBUFFER
pid	PID
pidi	PIDINFO
pmp	PMPARAM
pmres	PMRESULT
pqmdop	PQMOPENDATA
pqpdop	PQPOPENDATA
progc	PROGCATEGORY
proge	PROGRAMENTRY
progt	PROGTYPE
ptl	POINTL
ptri	POINTERINFO
pts	POINTS
qmsg	QMSG
qver	QVERSADATA
rcl	RECTL
rcl	RECTIS

Prefix	OS/2 Data Type
resc	RESULTCODES
rgb	RGB
rgnrc	RGNRECT
sbcd	SBCDATA
sel	SEL
sizfx	SIZEF
sizl	SIZEL
sizs	SIZES
smhs	SMHSTRUCT
stdata	STARTDATA
str8	STR8[8]
str16	STR16[16]
str32	STR32[32]
str64	STR64[64]
stdata	STATUSDATA
swctl	SWCNTRL
swp	SWP
swpus	SWPUSHORT
ti	TRACKINFO
tid	TID
ubtn	USERBUTTON
uch	UCHAR
ui	UINT
ul	ULONG
us	USHORT
vioci	VIOCURSORINFO
viofi	VIOFONTINFO
viain	VIOCONFIGINFO
vioint	VIOINTENSITY
viomi	VIOMODEINFO
vioos	VIOOVERSCAN
viopal	VIOPALSTATE
viopb	VIOPHYSBUF
vord	VORDER
wprm	WNDPARAMS
wpt	WPOINT
wrc	WRECT
xywin	XYWINSIZE

D

Restricted KBD, MOU, and VIO Calls

You have seen that there are several different types of programs that can be written to run under OS/2. The Presentation Manager session has many unique characteristics, and not all the base OS/2 APIs are supported in the Presentation Manager session. The list in this appendix contains those calls that may not be used under the Presentation Manager session.

In designing a program, you must be sure not to rely on any of these functions if your application is to run in a window, whether it be a “full-fledged” Presentation Manager application or a VIO-windowed application.

The following KBD calls are not allowed in applications that are to run in the Presentation Manager session.

KbdDeRegister

KbdRegister

The following MOU calls are not allowed in applications that are to run in the Presentation Manager session.

MouDeRegister

MouRegister

The following VIO calls are not allowed in applications that are to run in the Presentation Manager session.

VioDeRegister

VioGetFont

VioGetPhysBuf

VioGetState

VioModeUndo

VioModeWait

VioRegister

VioSavRedrawUndo

VioSavRedrawWait

VioScrLock

VioScrUnLock

VioSetFont

VioSetState

E

Include File Conditional Selection

Conditional Section Identifiers

This list contains all the values you can use to include specific sections of the OS/2 INCLUDE files. When you define a symbol, the INCLUDE files notice the definition and include headers. In many cases, defining a single value causes others to be defined by the INCLUDE files as a result. Such an example is INCL_BASE. This causes all the symbols associated with the base system to be defined and the relevant INCLUDE file sections to be included.

The procedure to cause a section of the headers to be included is to simply enter **#define** for the symbol prior to including the file OS2.H. An example is

```
#define INCL_WINCLIPBOARD    /* Define the symbol for the clipboard headers */
#include <OS2.H>              /* Include the OS/2 master header file      */
```

This is only an example; other items require definition to make this example functional. This procedure is shown for demonstration purposes only.

These INCL_ values may be defined in any order and in any combination as long as the definition is done prior to including OS2.H.

The list of INCL_ symbols and what they cause to be included is as follows:

#define:

INCL_AVIO
INCL_BASE

To include:

OS/2 AVIO headers and constants
The entire OS/2 Base

INCL_BITMAPFILEFORMAT	OS/2 Bitmap file format
INCL_DEV	OS/2 Device support
INCL_DEVERRORS	OS/2 Device error codes and functions
INCL_DOS	OS/2 DOS Kernel functions
INCL_DOSDATETIME	OS/2 Base date/time and timer functions
INCL_DOSDEVICES	Device specific, ring 2 functions
INCL_DOSERRORS	OS/2 Base error codes and functions
INCL_DOSFILEMGR	File management functions
INCL_DOSINFOSEG	InfoSegment (InfoSeg) functions
INCL_DOSMEMMGR	Memory management functions
INCL_DOSMODULEMGR	Module manager functions
INCL_DOSMONITORS	The OS/2 monitor functions
INCL_DOSNLS	National Language Support
INCL_DOSNMPIPES	The Named Pipes functions
INCL_DOSPROCESS	Process and thread management functions
INCL_DOSPROFILE	DosProfile functions
INCL_DOSQUEUES	Manager functions
INCL_DOSRESOURCES	Resource support
INCL_DOSSEMAPHORES	Semaphore APIs
INCL_DOSSESMGR	Session Manager functions
INCL_DOSSIGNALS	Signal functions
INCL_ERRORS	OS/2 Error codes and functions
INCL_FONTFILEFORMAT	OS/2 Font file format
INCL_GPI	All of the GPI functions and types
INCL_GPIBITMAPS	Bitmaps and Pel Operations
INCL_GPICONTROL	PS control functions
INCL_GPICORRELATION	Picking, Boundary, and Correlation APIs
INCL_GPIERRORS	GPI errors, defined if INCL_ERRORS is defined
INCL_GPILCIDS	Physical and logical fonts with Lcids
INCL_GPILOGCOLORTABLE	Logical color tables functions
INCL_GPIMETAFILES	Metafile functions
INCL_GPIPATHS	Paths and clipping with paths functions
INCL_GPIPRIMITIVES	Drawing primitives and primitive attributes
INCL_GPIREGIONS	Region and clipping with region functions
INCL_GPISEGEDITING	The segment editing via elements functions
INCL_GPISEGMENTS	The segment control and drawing APIs
INCL_GPITRANSFORMS	Transform and transform conversion functions
INCL_KBD	The Kbd APIs
INCL_MOU	The Mou APIs
INCL_ORDERS	OS/2 graphical order formats

INCL_PIC	OS/2 picture interchange
INCL_PM	All OS/2 Presentation Manager APIs and types
INCL_SAADEFS	SAA definitions from files
INCL_SHLERRORS	The shell error codes, defined if INCL_ERRORS is defined
INCL_SPL	The OS/2 spooler functions
INCL_SPLERRORS	This includes the spooler API error codes
INCL_SUB	This will include all the OS/2 Vio/Kbd/Mou APIs
INCL_VIO	The Vio APIs
INCL_WIN	OS/2 Window Manager functions
INCL_WINACCELERATORS	Keyboard accelerators
INCL_WINATOM	The Atom Manager APIs
INCL_WINBUTTONS	Button control functions
INCL_WINCATCHTHROW	WinCatch/WinThrow functions
INCL_WINCLIPBOARD	The clipboard functions
INCL_WINCOUNTRY	Country support
INCL_WINCursors	Text cursor functions
INCL_WINDIALOGS	Dialog box types and APIs
INCL_WINENTRYFIELDS	Entry field types
INCL_WINERRORS	Window error code definitions
INCL_WINFRAMECTLS	Frame controls (title bars & size border)
INCL_WINFRAMEMGR	Frame Manager functions
INCL_WINHEAP	Heap Manager APIs
INCL_WINHOOKS	Hook Manager functions
INCL_WININPUT	Functions for mouse and keyboard input
INCL_WINLISTBOXES	Listbox control support
INCL_WINMENUS	Menu control support
INCL_WINMESSAGEMGR	Message management functions
INCL_WINPOINTERS	Functions that deal with the mouse pointer
INCL_WINPROGRAMLIST	The Shell program list API
INCL_WINRECTANGLES	Rectangle functions
INCL_WINSROLLBARS	Scroll bar control support
INCL_WINSHELLDATA	Presentation Manager profile calls
INCL_WINSTATICS	Static controls
INCL_WINSWITCHLIST	The Shell switch list APIs
INCL_WINSYS	System values (and colors)
INCL_WINTIMER	Timer functions
INCL_WINTRACKRECT	The WinTrackRect() function
INCL_WINWINDOWMGR	General window management functions

F

Helper Macros

General Helper Macros

General helper macros are defined throughout the OS/2 INCLUDE files. There are macros for manipulating data pointers, ones for creating complex data types from simple ones, and still others for extracting the simple types given a complex type. This list of macros is useful for any type of program—Presentation Manager or otherwise.

The following are the most commonly used macros. They may be used to build and cast the data types necessary for the OS/2 functions.

```
/* MAKEP may be used to build a pointer from a selector and offset.      */
MAKEP(sel, off)                  /* MAKEP will create an untyped far pointer from */
                                /* a given selector and offset: "sel" and "off" */

/* These macros help split a pointer into its component selector and offset */
SELECTOROF(p)                    /* This will extract the selector from pointer, "p" */
OFFSETOF(p)                      /* This will extract the offset from pointer, "p" */

/* MAKETYPE is useful for casting data types */
MAKETYPE(v, type)                /* This will cast any variable, "v", to a data */
                                /* type, specified by "type" */
                                /* An example is MAKETYPE(var,ULONG), */
                                /* which will cast "var" to a ULONG type */
```

```

/* This macro gives to the byte offset of a field within a structure */
FIELDOFFSET(type, field) /* This macro will calculate the byte offset of */
                          /* a field in a structure of type "type" */

/* This set of macros will build a 32-bit value */
MAKEULONG(l, h)          /* Combines "l" and "h" to form a ULONG (32-bit) */
MAKELONG(l, h)           /* Combines "l" and "h" to form a LONG (32-bit) */

/* This set of macros will build a 16-bit value */
MAKEUSHORT(l, h)         /* Combines "l" and "h" to form a USHORT (16-bit) */
MAKESHORT(l, h)          /* Combines "l" and "h" to form a SHORT (16-bit) */

/* This set of macros extract high and low order components of */
/* 16- and 32-bit values */
LOBYTE(w)                /* This extracts the low order byte from a 16-bit value */
HIBYTE(w)                /* This extracts the high order byte from a 16-bit value */
LOUCHAR(w)               /* This extracts the low order byte from a 16-bit value */
HIUCHAR(w)               /* This extracts the high order byte from a 16-bit value */
LOUSHORT(l)              /* This will extract the low USHORT from a 32-bit value */
HIUSHORT(l)              /* This will extract the high USHORT from a 32-bit value */

```

Window-Specific Helper Macros

The following macros are designed to manipulate window-specific values such as message parameters (MPARAM). Each macro is defined in the OS/2 INCLUDE files and requires no data preparation.

```

/* These are the definitions of MPARAM and MRESULT, which are required */
/* when you work with windows and window procedures */

typedef VOID FAR *MPARAM;      /* An MPARAM is a far pointer */
typedef MPARAM FAR *PMPARAM;  /* A PMPARAM is a pointer to an MPARAM */
typedef VOID FAR *MRESULT;    /* An MRESULT is a far pointer */
typedef MRESULT FAR *PMRESULT; /* A PMRESULT is a pointer to an MRESULT */

/* The following macros may be used to build an MPARAM value from several */
/* data types. The macro name indicates the simple data type to be used */
/* in building the MPARAM */

MPFROMP(p)                  /* This makes an MPARAM from a pointer */
MPFROMHwnd(hwnd)            /* Makes an MPARAM from a window handle */
MPFROMCHAR(ch)              /* Makes a CHAR (ch) into an MPARAM */
MPFROMSHORT(s)              /* This makes a SHORT value (s) into an MPARAM */
MPFROM2SHORT(s1, s2)        /* This combines 2 SHORT values into one MPARAM */
MPFROMSH2CH(s, uch1, uch2)  /* Makes an MPARAM from a short and 2 CHARs */

```



```

MPFROMLONG(1)          /* This makes an MPARAM from a single LONG value */

/* The following macros may be used to extract standard data types from an
/* MPARAM value
PVOIDFROMMP(mp)        /* This casts an MPARAM to a pointer
HWNDFROMMP(mp)         /* This casts an MPARAM to a window handle
CHAR1FROMMP(mp)        /* Extracts the high order CHAR value from the high
                        /* order half of the given MPARAM
CHAR2FROMMP(mp)        /* Extracts the low order CHAR value from the high
                        /* order half of the given MPARAM
CHAR3FROMMP(mp)        /* Extracts the high order CHAR value from the low
                        /* order half of the given MPARAM
CHAR4FROMMP(mp)        /* Extracts the low order CHAR value from the low
                        /* order half of the given MPARAM
SHORT1FROMMP(mp)       /* This extracts the high order SHORT value from
                        /* the given MPARAM
SHORT2FROMMP(mp)       /* This extracts the low order SHORT value from
                        /* the given MPARAM
LONGFROMMP(mp)         /* This casts an MPARAM to a LONG value

/* These macros may be used to make an MRESULT value from standard types
MRFROMP(p)             /* This casts a pointer to an MRESULT value
MRFROMSHORT(s)         /* This casts a SHORT value to an MRESULT
MRFROM2SHORT(s1, s2)   /* Combines two SHORT values to build an MRESULT
MRFROMLONG(1)          /* Casts a LONG value to an MRESULT

/* These macros may be used to extract standard data types from an MRESULT
PVOIDFROMMR(mr)        /* This casts the given MRESULT to a pointer
SHORT1FROMMR(mr)       /* Extracts the high order SHORT from the given MRESULT
SHORT2FROMMR(mr)       /* Extracts the low order SHORT from the given MRESULT
LONGFROMMR(mr)         /* Casts an MRESULT value to a LONG

```

Index

- 80286 7, 13, 14, 16
- 80386 9, 16
- 8086 13
- 8088 5, 6
-
- Absolute address 69
- Accelerators 41, 42, 83, 85, 110, 111, 188, 189, 199, 218
- Action Bar 21, 40, 141, 212-219
- Addresses 99
- ALT-ESC 110
- Anchor block handle 98, 128, 129, 238
- API. *See* Application programming interface
- Application Data Transfer 323
- Application Programming Interface 12, 14, 24, 63-137
 - restricted calls 36
- Applications
 - AVIO 20, 37
 - DOS 14, 15, 24
 - FAPI 14, 15
 - graphics 20, 37
 - OS/2 14
 - Presentation Manager 20, 32, 50
 - VIO-windowed 20
 - windowed 34
- Arbitrary Units 241
- Arc 80, 248, 250
 - parameters 245
 - primitives 245
- Areas 83, 251
 - attribute—background color 254
 - attribute—background mix 254
 - attribute—foreground color 254
 - attribute—foreground mix 254
 - attribute—pattern set 254
 - attribute—symbol 254
 - parameters 251
- AVIO 20, 60, 76, 77, 78, 79
 - program 264, 351
-
- Background 31, 32, 72
- Basic Input/Output System 64, 72
- Basic Video Subsystem 68
- BA_ALTERNATE 251, 285, 287, 288, 290
- BA_NOBOUNDARY 251

- BA_WINDING 252, 285, 287, 288, 290
- Bezier curve 249
- Bindings 94
- BIOS *See* Basic Input/Output System
- Bitmap 80, 82, 85, 257, 303
 - data 304
 - editor 22
 - header 303
 - patterns 257, 303
- BM_SETCHECK 187, 190
- BM_SETHILITE 188
- BN_Clicked 172, 186–189, 205, 207
- BOOL 98
- Boot partition 73
- BS_AUTOCHECKBOX 189, 190
- BS_AUTORADIOBUTTON 186, 187
- BS_CHECKBOX 189, 190
- BS_DEFAULT 146
- BS_RADIOBUTTON 186, 187
- BSEDOS.H 96, 99
- BSEERR.H 96, 99
- BSE.H 96
- BSESUB.H 96, 99
- Build process 102
- Bus 62
- Buttons 135
- BVS *See* Basic Video Subsystem

- C language 94, 97, 127, 132, 138
- CBM_* values 198
- CBN_* values 198
- CBS_* values 198
- CF_DSPTEXT 326
- CFLSELECTOR 326
- CHAR 98
- Character 255
 - attribute—Angle 256
 - attribute—Box 256
 - attribute—Shear 256
 - mode—CM_MODE1 256
 - mode—CM_MODE2 256
 - mode—CM_MODE3 256
- Check box 46, 47, 171, 180, 186, 189, 190
- Child window 34, 139
- Class Style 130, 144–145 152, 153
- Click and drag 176
- Client window 131, 143, 146, 148, 151, 185, 219
- Clipboard 20, 23, 47–48, 89, 90, 205, 213, 323–328
 - application interface 324
 - copy 47, 325, 328
 - cut 47, 325, 328
 - data format 326, 327
 - paste 47, 48, 327
 - user interface 324
- Clipping 88
- Closing an area 251, 254
- CMD.EXE 13
- CMDSRC_* values 169
- CMDSRC_ACCELERATOR 188, 189
- CMDSRC_MENU 188, 189
- CMDSRC_PUSHBUTTON 188, 189
- COBOL 22, 24, 94
- Code segments 64
- Codepage 275
- Combo box 47, 180, 186, 197
- COMMAND.COM 13
- Command Reference 24
- Common Programming Interface 20, 77
- Common User Access 20, 77
- Compatibility box. *See* DOS box
- Compiler 70, 71
- Compiler switches 99
- Conditional sections 95
- Contextual help 88
- Control flags 131
- Control Panel 51–52, 53, 54, 181
- Controls 141–144, 181, 185, 219
- Coordinate size 237
- Coordinate Space 296
- Copy 89, 90
- Copying Segment Data 293
- CPI. *See* Common Programming Interface
- Creating a Bitmap 304
- Creating a Dynamic Link Library Stub 340
- Creating a Font Library 342
- Creating a PS 238
- Creating a resource library 343

- Creating a Segment 283
- Creating and Using Window Templates 370
- Creating Markers 278
- Creating Patterns 279
- CS_SIZEREDRAW 353, 365
- CS_* flags 144-145
- CS_SYNCPOINT 153, 173
- CUA. *See* Common User Access
- current position 257, 268, 270
- Cut 89, 90

- Data exchange 89
- Data segment 100
- Data types 95
- DCTL_DYNAMIC 294
- DEF file 100, 101, 102
- Default Processing 84, 117
- Demand loading 12
- Dependency 103
- Descriptor 64, 65
- Descriptor Table 64-66
- Desktop Manager 19, 23, 24, 33, 34, 54, 82
 - cascading windows 48, 49
 - closing all running programs 49
 - saving window arrangement 48-49
 - shutting down the system. *See* Shutdown
 - tiling windows 48, 50
- Desktop window 33, 130, 138, 139, 147, 153, 156, 184
- DEV 80
- Dev call 79, 81
- DevCloseDC 306, 311
- DevEscape 313, 314, 317
- Device Context 78, 79, 88, 97, 227, 229, 238, 239, 297, 365
- Device Drivers 17-18, 63, 64, 66, 67, 68, 69, 72, 77, 79, 80
 - kernel 19, 22
 - Presentation Manager 22
- Device independence 66, 68, 78, 81, 88
- Device mapping 88
- DevOpenDC 229, 230, 231, 305, 308, 312, 315, 316, 317

- DEVOPENSTRUCT 230, 312, 315
- Dialogs 76, 82, 85, 91, 94, 180
- Dialog box 46-47, 180-189, 205, 213, 221
 - modal 182, 183 207-209
 - application modal dialog 183, 184, 185
 - system modal dialog 183
 - modeless 182, 207
- Dialog editor 22
- Dialog Manager 22, 23, 25
- Dialog procedure 182, 186, 205, 206, 208
- Dialog Tag Language 23
- Dialog Tag Language Compiler 23
- Direct I/O 63, 81, 82
- Direct Printing 311
- Disable interrupts 63
- Dispatcher 59
- DLL *See* Dynamic Link Library
- DM_DRAW 282
- DM_DRAWANDRETAIN 283
- DM_RETAIN 283
- DOS 63, 64, 66, 67, 72, 76, 102, 104, 105
 - 1.0 5-6
 - 1.1 6
 - 2.0 6-7
 - 2.1 7
 - 3.0 7-8
 - 3.1 7, 8
 - 3.2 8-9
 - 3.3 9-10, 13, 15, 23, 76
 - 4.0 10, 22, 23, 24, 31, 76
 - applications, 14, 15, 24, 32, 37
- DOS box 13, 24, 31-32, 37, 76, 101
- DOS command prompt 17, 31
- DOS mode 13, 15. *See also* DOS box
 - addressability 13
 - command processor 13
- DOS shell 10
- Dos Call 70, 81, 82, 128
- Dos Compatibility Mode 75, 76
- DosAllocSeg 79, 166, 273, 274, 313
- DosBeep 165, 167, 170, 171
- DosClose 306, 311
- DosCreateThread 70, 79, 167

- DosDevIOCtl 79
- DosExit 116
- DosFreeSeg 79
- DosKillProcess 79
- DosOpen 306, 311
- DosWrite 306, 311
- Double click 174, 175
- Draw and Retain Mode 283
- Draw Mode 282
- Drawing
 - boxes 260
 - circles 245
 - curves 245
 - ellipses 245
 - full arcs 246
 - lines 247
 - modes 282
 - multiple arcs 248
 - partial arcs 247
 - point arcs 247
 - segments 287
 - text 263, 265, 266, 267
 - wavy lines 245
- DS register 100
- DT_QUERYEXTENT 267
- DT_WORDBREAK 267
- DTL. *See* Dialog Tag Language
- DTLC. *See* Dialog Tag Language
 - Compiler
- Dynalink 70
- Dynamic Data Exchange (DDE) 83, 90, 91, 112, 323, 329
- Dynamic Link Library 16, 17, 70–74, 79, 86, 100, 131, 142, 143, 340
- Dynamic linking 12, 16, 70, 71, 100
 - load time 16
 - run time 16
- EM_SETINSERTMODE 197
- EM_* values 195
- EN_* values 195
- Enhanced graphics display 66
- Entry field 147, 171, 178, 180–204, 335, 337
 - insert mode 197
 - multiline. *See* Multiline edit control
 - replace mode 197
 - single line 46
- Entry point 71
- ES_* values 195
- Event driven 105, 113, 123, 164
- Executable program 94
- EXE-header 71, 101
- Extended attributes 73
- Extended partition 73
- Extended session 60
- extended VIO attributes 351, 365
- EXTERN 70, 71
- External reference 70, 71, 100
- Family application programming interface
 - applications 14, 15, 32, 76
- FAPI. *See* Family application programming interface
- Far call 70, 71
- FAT 73
- FCF_HORIZSCROLL 372
- FCF_MAXBUTTON 372
- FCF_MENU 213, 215.372
- FCF_MINBUTTON 372
- FCF_SHELLPOSITION 372
- FCF_SIZEBORDER 372
- FCF_STANDARD 214
- FCF_SYSMENU 372
- FCF_TASKLIST 372
- FCF_TITLEBAR 372
- FCF_VERTSCROLL 372
- FCF_* flags 149–150
- FC_* flags 156–157
- FID_HORIZSCROLL 359, 360
- FID_MENU 213, 221
- FID_VERTSCROLL 360
- FID_* values 159
- File
 - cache 73
 - handle 69
- File Manager 23. *See also* File system
- File system 21, 52, 54, 73
 - refresh option 51
- Focus 35, 37, 155, 195, 205

- Focus change flags 156
- Focus window 79, 112, 155, 157, 179
- Font 83, 85, 255, 340, 342
 - attribute structure 275
 - classes 255, 273
 - editor 22
 - face name 255, 275
 - fixed 21
 - group 255, 273
 - image 255
 - logical name 276
 - outline 255
 - private 255
 - proportional 21
 - public 253
 - raster 255
 - selection 256, 273, 275
 - system 21
 - usage considerations 257
 - vector 255
- Foreground 31, 32, 39, 72
- FORMAT 73
- Fortran 22, 24, 94
- Frame control flags 149
- Frame window 131, 141, 148, 151, 181, 184, 185, 219, 221
- Full screen 60, 124
 - mode, 31, 32, 37
 - text 76, 81, 100
- Function Prototype 70
- GDT. *See* Global Descriptor Table
- General Protection Fault 63, 71
- Get message loop 114, 116, 131
- Global Descriptor Table 64, 65
- GPI 78, 79, 83, 227
- Gpi call 81, 82, 95, 100
- GpiAssociate 238, 287, 292, 309, 310, 316, 317, 318
- GPIA_ASSOC 239
- GPIA_NOASSOC 239
- GpiBeginArea 251, 285, 287, 288, 290
- GpiBeginPath 260
- GpiBitBlt 306, 316, 317
- GpiBox 252, 253, 254, 260
- GpiCallSegmentMatrix 289, 294
- GpiCharString 268, 271, 272, 283, 293
- GpiCharStringAt 133, 173, 269, 306, 309, 314, 317
- GpiCharStringPos 269, 270, 271, 272
- GpiCharStringPosAt 269, 271
- GpiCloseSegment 284, 287, 289, 292
- GpiCreateBitmap 304, 305, 316
- GpiCreateLogFont 275, 277, 279, 314
- GpiCreatePS 242, 305, 309, 312, 315
- GpiDeleteBitmap 318
- GpiDeleteMetafile 310
- GpiDeleteSetID 280
- GpiDestroyPS 318
- GpiDrawChain 289, 294
- GpiDrawFrom 294
- GpiDrawSegment 287, 292
- GpiEndArea 251, 285, 287, 288, 291, 292
- GpiEndPath 260
- GpiFullArc 246
- GpiGetData 293
- GpiLabel 289, 290, 291
- GpiLine 247, 258, 283, 285, 286, 291
- GpiLoadBitmap 170, 279
- GpiMarker 261, 278
- GpiMove 173, 246, 247, 249, 250, 251, 253, 254, 258, 268, 269, 284
- GpiOffsetElementPointer 293
- GpiOpenSegment 284, 287, 290
- GpiPartialArc 247
- GpiPlayMetafile 308, 309, 310
- GpiPointArc 247
- GpiPolyFillet 248, 284, 287, 290, 291, 292
- GpiPolyLine 258, 260, 288
- GpiPolyMarker 262, 278
- GpiPolySpline 250
- GpiPutData 293
- GpiQueryBitmapBits 306
- GpiQueryCharStringPos 271
- GpiQueryCharStringPosAt 271
- GpiQueryFonts 273, 274, 312, 313
- GpiQueryMetafileBits 308
- GpiQueryTextBox 271
- GpiSaveMetafile 310
- GpiSetArcParms 245

- GpiSetBackColor 284, 287, 290
- GpiSetBitmap 170, 304, 305, 307, 316, 317, 318
- GpiSetBitmapID 280
- GpiSetCharAngle 271, 272
- GpiSetCharBox 272
- GpiSetCharDirection 271
- GpiSetCharSet 276, 277
- GpiSetColor 268, 269, 284, 285, 287, 288, 290, 291, 293
- GpiSetCurrentPosition 258
- GpiSetDrawingMode 282, 283, 284, 287, 290
- GpiSetEditMode 293
- GpiSetElementPointer 293
- GpiSetElementPointerAtLabel 293
- GpiSetLineEnd 260
- GpiSetLineJoin 260
- GpiSetLineWidthGeom 260
- GpiSetMarker 262, 278, 279
- GpiSetMarkerSet 278, 279
- GpiSetMetafileBits 308
- GpiSetModelTransformMatrix 298
- GpiSetPageViewport 300
- GpiSetPatternSet 280
- GpiSetSegmentAttrs 294
- GpiSetSegmentTransformMatrix 298
- GpiSetViewingTransform 300
- GpiStrokePath 260
- GpiUnloadFonts 278
- Graphics 20
 - attributes 20
 - bounds 20
 - clipping 20
 - correlation 20
 - element 289
 - element offset 293
 - engine 77, 79, 80
 - nonretained 20
 - order 289
 - primitives 20, 82, 83, 227, 244, 284
 - processing 20
 - retained 20
 - transformation 20
- Graphical User Interface 75
- HAB 98
- Handle 69, 86, 97
- Hard-error handler 19, 63
- Hardware independence 80
- Header files 70, 95, 98, 177
- Headers 70, 95, 123, 128
- Heap 101
- High Performance File System 22, 23, 25, 49, 73
- HK_HELP 88
- HMQ 98
- HPFS. *See* High Performance File System
- Hook 75, 83, 86, 88
 - application 86
 - chain 86
 - procedure 86
 - system 87
- Hot keys 31, 39
 - Alt-Esc 39, 40
 - Alt-Tab 39
 - Ctrl-Esc 39
- HWND 98
- HWND_DESKTOP 130, 184, 209
- Icon 34, 85, 94, 108, 128
 - editor 22
 - selection 21
 - sizing 43
- Image 257
 - attributes 257
- Include files 22, 95, 97, 98, 99, 104, 128
- INCL_* 95, 104
- Indirection 64, 69
- Information Presentation Facility 22, 23-24, 25
- Information Presentation Facility Compiler 24
- Installable File System (IFS) 73
- Installation procedure 66
- INT architecture 76
- INT3 coded in C 348
- Interprocess communication 12, 17, 32, 70, 76
- I/O control packet (IOCTL) 79
- I/O Privilege level (IOPL) 63, 64

- I/O services 73
- IPC. *See* Interprocess communication
- IPF. *See* Information Presentation Facility
- IPFC. *See* Information Presentation Facility Compiler

- KBD 79
- Kbd call 81
- KC_* flags 176
- KC_KEYUP 359, 366
- Kernel 17-19, 62, 63, 68, 69, 72, 79, 80
- Keyboard keys 82, 83
 - Alt 41
 - Alt-Spacebar 40
 - Ctrl-PgDn 45
 - Ctrl-PgUp 45
 - F10 41
 - Shift-Esc 40
 - Shift-F7 45
 - Shift-F8 45
 - using 39-46. *See also* Hot keys
- Keyboard Subsystem 17, 68

- Languages 25
 - COBOL 22, 24, 94
 - FORTRAN 22, 24, 94
 - Pascal 23
- LDT *See* Local Descriptor Table
- LIB file 100
- Line 80, 83
 - multiple lines 258
 - primitives 257
 - width 260
- LINK 70
- Linker 86, 100-102
- Listbox 46, 47, 171, 180, 181, 186, 190-198, 206
- LM_INSERTITEM 181, 182, 192, 193
- LM_QUERYITEMHANDLE 194
- LM_SETITEMHANDLE 194
- LM_* values 192-193
- LN_ENTER 194
- LN_KILLFOCUS 194
- LN_SCROLL 194
- LN_SELECT 181, 194
- LN_SETFOCUS 194
- Loader 71
- Local Descriptor Table 64-66, 71
- Local identifier 276
- LONG 98
- Long file names 73
- LS_MULTIPLESEL 191
- LS_NOADJUSTPOS 191
- LS_OWNERDRAW 191-194

- MAKE utility 102, 103, 104
- Makefile 103
- Mapping 69
- Marker
 - attributes 262
 - primitive 261
- MBID_* values 210-211
- MB_* values 209-210
- Memory
 - address 64, 65, 69
 - addressability 11, 13, 24
 - compaction 64, 66
 - Device Context 232
 - management 12, 13, 16, 64, 68, 70, 73
 - models 100
 - overcommitment 13, 16, 64
- Menu 76, 79, 82, 85, 94, 135, 188, 198, 212-221, 373
 - attributes 212
 - ID 213
 - item 215-223
 - item selection 40-42
 - pull-down 21, 40 212, 213, 216, 219
 - system 40
 - template 212, 213
- Message 75, 76, 82, 84-88, 90, 105-117, 132, 133, 134, 163
 - architecture 80, 105, 113, 123
 - handling 108, 117, 132, 133, 163
 - ID 84, 106, 116, 117, 132
 - keyboard 110, 176, 179
 - parameters 107, 110, 117
 - passing mechanism 76
 - posting 108, 160-161, 165, 168

- queue 84, 86, 88, 97, 106–109, 123, 128, 129, 132, 133, 160, 161
- queue handle 98, 128
- return values 167
- sending 106, 108, 160–161, 168, 181
- structure 106
- time guideline 166
- user input 155, 163, 185, 205
- Message box 46, 47, 208, 211, 346, 347
- Message router 110, 111
- Metafile 82, 308
- Metafile Device Context 231
- Metric units 240
- MIA_DISABLED 223
- MIA_* values 218
- MIS_* values 217, 218
- MLN_* values 204
- MLS_* values 204
- MM_SELECTITEM 219
- MM_SETITEMATTR 223
- MM_* messages 222–223
- Modality 182, 205
- Module definition file 100, 101, 128
- Monitor 81, 86, 88
- MOU 79
- Mou call 81, 82
- Mouse 79, 83
 - action button 38
 - capture 157, 158
 - click 38
 - DOS box 32
 - double click 38
 - dragging 39
 - input 82
 - message 110, 111, 174–178
 - pointer 38
 - subsystem 17, 68
 - using 38–46
- Multiline edit control 47, 204
- Multitasking 11, 13, 16, 24, 59, 60, 64, 66, 69, 89, 108
 - parallel 15
 - serial 15
- Naming conventions 95, 96, 124
- Notification message 141, 143, 163, 171, 186, 187, 195–200, 204, 211
- NOTWINDOWCOMPAT 100
- OBJ file 86, 100, 102, 103
- Object 75, 82, 83, 84, 88, 94, 136, 137
- Object-Oriented Programming 83, 135, 136, 137, 161
- OD_DIRECT 232
- OD_MEMORY 315
- OD_METAFILE 231
- OD_QUEUED 233
- Offset 64, 65, 69
- Online documentation 22, 23, 24, 25, 73
 - obtaining online help 45–46
- OOP *See* Object-Oriented Programming
- Operating environments 31–37
 - DOS compatibility box. *See* DOS box
 - OS/2 full-screen mode. *See* Full-screen mode
 - Presentation Manager user shell 31
- OS/2
 - advantages over DOS 11
 - applications 14, 32
 - OS/2 1.0 10, 11–18, 19, 24, 32, 36, 37, 59, 67
 - OS/2 1.1 18–22, 24
 - OS/2 1.2 22–25
 - OS/2 command prompt 17, 32, 35–36
 - OS/2 mode 13, 15, 32
 - addressability 13
 - command processor 13
 - OS2DEF.H 96, 99
 - OS2.H 95, 96, 104, 128
 - OS2.INI 91
 - Owner-owned relationship 138, 141, 184
- Parent Window 139
- Parent-child relationship 138, 184
- Paste 90
- Paths 260
- Patterns
 - Bitmap 279
 - Font 280

- Pel 240, 257, 267
- Performance 80
- Picture utilities 23
 - PICICHG 23
 - PICPRINT 23
 - PICSHOW 23
- Pipes 17, 76, 323
- PMAVIO.H 96, 99
- PMBITMAP.H 96
- PMDEV.H 96
- PMFONT.H 96
- PMGPI.H 95, 96, 99
- PMORD.H 96
- PMPIC.H 96
- PMSHL.H 96, 99
- PMSPL.H 96
- PMWIN.H 95, 96, 99
- PM.H 96
- Pointer 83, 85, 97
- Portability 82
- Presentation drivers 77, 79, 80, 155
- Presentation Manager
 - characteristics 19
 - enhancements in OS/2 1.2 22-23
- presentation page 235, 237, 239, 296, 297, 299
- Presentation services 80
- Presentation Space
 - allocation 238
 - association 88, 235
 - AVIO 351
 - cached micro 237
 - character 79, 83, 88, 97
 - handle 132, 133, 173, 235
 - micro 236
 - normal 236
 - size 239, 242, 297, 365
 - VIO 264
- Prf call 82
- PrfAddProgram 91
- Primitives 227, 244
- Print Manager 52-53 54. *See also* Spooler
- Printing 310
 - graphics 310
 - queued 314
- Priority 62
- Private class 143, 147
- Privilege level 61-63
- Process 60, 61, 64, 69, 72, 79
 - ID 66, 73
- Processor State 61
- Program isolation 74
- Program Selector 81
- Program installation 48, 49, 50, 54
- Programming Reference 24
- Program Selector 10, 17, 18, 19
- Protect mode 13, 76. *See also* OS/2 mode
- Protection 11, 15, 60, 61, 62, 66
- Protection rings 61, 62
- Protection violation 63, 65, 66, 69
- PSTAT 73
- Public classes 130, 142, 143, 147, 151
- Pushbutton 46, 47, 146, 180, 181, 186, 188, 189, 207, 208
- QF_PRIVATE 273, 312, 313
- QF_PUBLIC 273, 312, 313
- Querying profile data 233
- Queue 17, 60, 76, 84, 86, 107-116
- Queued Printer/Plotter DC 233
- Radio button 46, 47, 180, 186, 187
- RC file 85, 86, 128, 148
- Real mode 13. *See also* DOS mode
- Reference record 71
- Registers 61
- Rendering 325
 - delayed 325, 327
 - immediate 325
- Replacing elements 292, 293
- Request packet 68, 72
- RES file 86, 102
- Resolution 265
- Resource 82, 85, 94
- Resource Compiler 22, 85, 86, 94, 102, 104, 213
- Resource file 148, 213, 216, 218
- Resource library 340, 345
- Restricted APIs 76
- Retain mode 283

- Return chain 334, 337-339
- Ring 0 61-63, 68
- Ring 1 61, 63
- Ring 2 61, 63
- Ring 3 61, 63, 64
- Rotation 301

- SAA. *See* Systems Application Architecture
- SBM_SETPOS 201, 203, 364
- SBM_* values 203
- SB_* values 199-200
- SB_ENDSCROLL 359, 360, 361
- SB_LINEDOWN 201, 360
- SB_LINELEFT 202, 359, 361
- SB_LINERIGHT 202, 359, 361
- SB_LINEUP 201, 203
- SB_PAGEDOWN 202, 360
- SB_PAGELEFT 361
- SB_PAGERIGHT 361
- SB_PAGEUP 201, 360
- SB_SLIDERPOSITION 361
- Scaling 301
- Scheduler 12, 16, 59
- Screen buffer 81
- Screen device context 231
- Screen group 60, 79, 81
- Scroll bar 21, 46, 135, 151, 180, 186, 190, 198, 199, 200, 204
 - horizontal 44, 45
 - vertical 44, 45
- Scrolling 76, 79
 - horizontal 359-362, 366
 - keyboard 359, 360, 366
 - vertical 359, 360, 363, 364, 366
- Security 61
- Segment
 - graphics 282-284, 289, 296
 - attributes 293
 - calling 287
 - closing 284
 - compound element 289
 - element 289
 - label 289, 290
 - opening 284
 - data or code 65, 69, 71, 72, 74
 - not present fault 66
 - swapping 16
- Segmentation Violation 63
- Segmented memory model 64
- Selector 64-66, 69, 74
- Semaphore 17, 60, 73, 80
- Session 13, 19, 31, 60, 61, 76, 81, 83, 86, 113
 - DOS mode 19
 - extended 19, 33, 35-37
 - full-screen 19, 32, 35, 36
 - ID 66, 72, 88
 - Presentation Manager 19, 32, 35, 36-37
 - protect mode 19
 - VIO-windowed 19, 35-36, 37
- Shared memory 17, 90, 323
- SHORT 98
- Shutdown 49
- Sibling window 35, 139, 154
- Signals, 17
- Single Input Queue 109, 110, 176
- Sizing 76, 79
- Spl call 82
- Spline 249
- Spooler 82
- Spooler, 19
- SPTR_WAIT 166
- Stack 61, 101
- Static library 71
- Static linking 16, 71, 100
- Strategy routine 68
- String 80, 85
- Structures 95
- STUB statement 100
- Stub dynamic link library 340
- Subclass procedure 333-339
- Subclassing 146, 167, 332, 334, 336-339
 - Multiple 338-339
- Submenu 212, 213, 215, 216, 218, 219, 221, 223
- Subsystems 17, 63, 66-72, 78, 79, 81, 82
- Swapping 64
- Switch statement 117, 132, 165, 169, 171, 177, 220
- SWP_* flags 140-141
- Symbol definitions 95

- Symbolic debugging 349
- System editor
 - OS/2 1.1 19
 - OS/2 1.2 22, 24
- System extensions 67
- System hook 86
- System Initialization 63, 68
- System services 69, 78
- Systems Application Architecture 20–21, 77, 129, 142, 148
- Target file 103
- Task 107
- Task List 19, 39, 50–51, 54, 91
- TCB. *See* Thread Control Block
- Text
 - character control 268, 269, 270
 - justification 267, 269
 - string control 268, 269
- Thread 59, 60, 61, 64, 70, 79, 80, 84, 86, 107–116, 133, 142, 160, 161, 166
 - context 61
 - switching 62
- Thread Control Block 61
- Title Bar 42, 212
- Tracking rectangle 325
- Transformation Matrix 300
- Transforms 296
 - device space 300
 - model 298
 - segment 298
 - viewing 299
- Translation 78, 80, 81, 88, 89, 300
- Trap D 63
- ULONG 98
- Unsubclassing 333, 339
- Update region 153, 154
- User input 82
- User shell 32–37, 39, 48–54
- USHORT 98
- Variable prefixes 97, 98
- Video subsystem 17, 72, 79
- Viewport 300
- VIO *See* video subsystem
- Vio call 81, 82
- Vio handle 264
- VIO popup 19, 21
- VIO Windowable 76, 79, 100
- VioAssociate 264, 354
- VioCreatePS 264, 354
- VioDestroyPS 264, 354
- VioGetDeviceCellSize 358, 366
- VioReadCharStr 357, 362
- VioSetCurPos 356
- VioSetOrg 361, 362
- VioWrtCharStr 264
- VioWrtNCell 355, 357
- VioWrtTTY 124, 132, 263
- VIO-Window 60, 76
- Virtual address space 13
- Virtual device 72
- Virtual Machine 60
- Virtual memory 16
- Viz-region 153
- VK_DOWN 360
- VK_LEFT 359
- VK_RIGHT 359
- VK_UP 360
- WC_BUTTON 146, 186, 188, 189
- WC_COMBOBOX 197
- WC_ENTRYFIELD 178, 194
- WC_FRAME 142, 143, 148
- WC_LISTBOX 190
- WC_MENU 175, 216
- WC_MLE 204
- WC_TITLEBAR 151
- WC_* 129, 130
- Win call 82, 95
- WinBeginPaint 133, 153, 173, 174, 237
- WinCloseClipbrd 326, 327, 328, 329
- WinCreateMsgQueue 128, 129
- WinCreateStdWindow 128, 130, 135, 148, 149, 151, 214, 343, 354, 370
- WinCreateWindow 82, 128, 135, 147–151, 191, 214, 344, 370

- WinDdeInitiate 91, 329
- WinDdePostMsg 329, 330
- WinDdeRespond 329
- WinDefAvioWindowProc 358
- WinDefDlgProc 182, 188, 205–207
- WinDefWindowProc 84, 115, 116, 117, 123, 132, 133, 164, 168, 175, 182, 205, 220, 334, 335, 338, 365
- WinDestroyMsgQueue 129, 354
- WinDestroyWindow 129, 207
- WinDismissDlg 206–208
- WinDispatchMsg 111, 115, 129, 131, 160, 354
- WinDlgBox 184, 205, 343
- Window 20, 33–35, 39–40, 53, 75
 - active 35
 - basic 79, 146, 147
 - class 130, 132, 135, 137, 138, 142, 146
 - creation 82
 - destruction 82
 - device context 229, 231, 242, 263
 - error handling 82
 - handle 94, 98, 106, 116, 128, 137, 151, 157–160, 182, 213
 - hierarchy 34, 35
 - ID 159, 160
 - main 34
 - manipulation 82
 - maximized 43
 - minimized 42–44
 - moving 20, 39, 42
 - procedure 88, 100, 101, 111–178, 182, 186, 205, 206, 219, 332–339
 - rectangle 153, 154
 - relationships 135, 138
 - restored 43–44
 - scrolling 44–45
 - sizing 20, 39, 42–44
 - standard 53, 131, 146, 148, 212, 213
 - styles 144, 152, 153
 - templates 370, 371, 372
 - text drawing 265, 271
 - user interaction 40–47
- WINDOWAPI 100
- WINDOWCOMPAT 100
- Window-compatible 79
- WinDrawBitmap 170
- WinEndPaint 133, 153, 173, 174
- WinFillRect 133, 170–173
- WinFocusChange 155–157
- WinGetLastError 170, 276
- WinGetMsg 111, 114, 115, 116, 129, 131, 153, 161, 354
- WinGetPhysKeyState 82
- WinGetPS 170–172, 237, 305, 315
- WinInitialize 82, 128, 129
- WinInvalidateRect 153, 154, 173
- WinInvalidateRegion 153, 154
- WinLoadAcceleratorTable 343
- WinLoadDlg 183, 205, 343, 370, 371
- WinLoadMenu 343, 371, 372
- WinLoadPointer 343
- WinLoadString 343
- WinMessageBox 170, 208–209, 347
- WinOpenClipbrd 326, 327, 328, 329
- WinOpenWindowDC 229, 231, 242, 355
- WinPeekMsg 111, 13, 161
- WinPostMsg 109, 111, 161, 170, 181, 201, 202
- WinProcessDlg 184
- WinQueryClipbrdData 327
- WinQueryProfileString 233
- WinQueryWindow 371
- WinQueryWindowRect 265, 266, 267, 271, 358
- WinQueryWindowText 197
- WinRegisterClass 128, 130, 143, 146,, 332, 353
- WinReleaseCapture 111, 157
- WinReleasePS 170–172, 316
- WinScrollWindow 203
- WinSendDlgItemMsg 182, 193
- WinSendMsg 109, 160, 161, 181, 182, 188, 193, 221, 223, 356, 357, 366
- WinSetCapture 111, 157
- WinSetClipbrdData 325, 327, 328, 329
- WinSetClipbrdOwner 327
- WinSetFocus 155–157
- WinSetHook 86
- WinSetOwner 151
- WinSetParent 139
- WinSetWindowPos 139, 140, 192, 355, 365

- WinShowWindow 183, 193
 - WinSubclassWindow 332–336
 - WinTerminate 129, 132, 354
 - WinValidateRect 153
 - WinValidateRegion 153
 - WinWindowFromID 159, 160, 182, 213, 219, 221–223, 356, 357, 362
 - WM_BUTTON1DOWN 107, 111
 - WM_BUTTON2DOWN 165
 - WM_BUTTON* 174–175
 - WM_CHAR 110, 111, 117, 132, 133, 155, 176–178, 200, 359, 365, 366
 - WM_CLOSE 132, 133
 - WM_COMMAND 110, 111, 168–171, 182, 188, 189, 207, 212–223, 356, 365
 - WM_CONTROL 141, 143, 171, 172, 173, 181, 182, 187, 189, 194, 195, 203–205, 210
 - WM_CREATE 106, 151, 152, 182, 355, 365
 - WM_DDE_ACK 330, 331
 - WM_DDE_ADVISE 331
 - WM_DDE_DATA 330, 331
 - WM_DDE_EXECUTE 331
 - WM_DDE_INITIATE 330, 331
 - WM_DDE_INITIATEACK 330, 331
 - WM_DDE_POKE 331
 - WM_DDE_REQUEST 330, 331
 - WM_DDE_TERMINATE 331
 - WM_DDE_UNADVISE 331
 - WM_DRAWITEM 141
 - WM_ENABLE 181
 - WM_HELP 209, 211
 - WM_HITTEST 110
 - WM_HSCROLL 199–203, 359, 365, 366
 - WM_INITDLG 168, 182, 188, 205, 207
 - WM_INITMENU 141
 - WM_MENUSELECT 219
 - WM_MOUSEMOVE 82, 157
 - WM_MOVE 151
 - WM_PAINT 107, 112, 131, 133, 152, 153, 154, 173, 174, 237, 359, 365, 366
 - WM_QUERYWINDOWPARAMS 197
 - WM_QUIT 116, 131, 133, 206
 - WM_RENDERALLFMTS 327
 - WM_RENDERFMT 327, 328
 - WM_SIZE 151, 365, 358, 366
 - WM_TIMER 112
 - WM_VSCROLL 199–203, 359, 365, 366
 - WS_CLIPCHILDREN 153, 154
 - WS_CLIPSIBLINGS 154
 - WS_MINIMIZED 152
 - WS_PARENTCLIP 153
 - WS_SYNCPAINT 138, 152–154
 - WS_VISIBLE 147, 152, 294
 - WS_* flags 144
- Z-order 35, 48, 110, 139, 154

A complete programming guide for OS/2's
new standard interface ...

OS/2TM PRESENTATION MANAGER PROGRAMMING

Here is expert guidance on programming the Presentation Manager, IBM's new standard interface for the OS/2 operating system. Written by key members of the Presentation Manager development team, *OS/2 Presentation Manager Programming* presents thorough, insider coverage of the interface's capabilities and features, including:

- The complete range of programming concepts, with information on building applications, message architecture, and program structure
- A comprehensive programming guide with numerous examples written in C
- Essential guidance on window programming and on the Graphics Programming Interface (GPI)

OS/2 Presentation Manager Programming features discussion of more advanced

topics, including application data transfer, templates, and advanced VIO programming, debugging, font creation, and menu management.

With its thorough, jargon-free presentation, along with the insider's perspective it provides, *OS/2 Presentation Manager Programming* is both an excellent learning tool and handbook on programming in this challenging new environment.

PAUL W. CHEATHAM, DAVID E. REICH, and ROBERT F. G. ROBINSON are members of IBM's OS/2 development team.

Cover Design: Laurie Angel-Sadis

Illustration: Stanislaw Fernandes

John Wiley & Sons, Inc.
Professional, Reference and Trade Group
605 Third Avenue, New York, N.Y. 10158-0012
New York • Chichester • Brisbane • Toronto • Singapore

SC28-2700-00

